New Techniques to Improve Network Security

Tesi di Dottorato

Matteo Avalle

Dottorato di Ricerca in Ingegneria Informatica - XXVI ciclo

Tutore:
Riccardo Sisto

Dipartimento di Automatica e Informatica (DAUIN)
Politecnico di Torino
Italy

Submitted: February 14, 2014

Formal Methods, Parallel Programming and Distributed Architectures: New Weapons to Enforce Network Security

Short abstract:

With current technologies it is practically impossible to claim that a distributed application is safe from potential malicious attacks. Vulnerabilities may lay at several levels (criptographic weaknesses, protocol design flaws, coding bugs both in the application and in the host operating system itself, to name a few) and can be extremely hard to find. Moreover, sometimes an attacker does not even need to find a software vulnerability, as authentication credentials might simply "leak" ouside from the network for several reasons. Luckily, literature proposes several approaches that can contain these problems and enforce security, but the applicability of these techniques is often greatly limited due to the high level of expertise required, or simply because of the cost of the required specialized hardware.

Aim of this thesis is to focus on two security enforcment techniques, namely formal methods and data analysis, and to present some improvements to the state of the art enabling to reduce both the required expertise and the necessity of specialized hardware.

Keywords: Network Security, Formal Methods, Parallel Computing, Distributed Computing

Contents

Resumé:
How can we make sure that a network is "secure"? It's impossible to guarantee an absolute security, but fortunately there are techniques able to provide an high level of confidence about each security property: some approaches act at design level, by using formal languages to model communication protocols and mathematically prove or disprove their properties. Other approaches, instead, focus on constantly monitoring the network to track and block suspect behaviors.

Both these approaches, however, are costly: on one hand, a high level of expertise is required to properly use formal languages to build meaningful security proofs; for what concerns the traffic monitoring, instead, a main problem regards performance. Even in the assumption that an analytic is powerful enough to track all the possible malicious behaviors, in fact, being able to apply such a complex algorithm to a huge amount of network traffic may require powerful, special-purpose, extremely expensive hardware.

Goal of this thesis is to mitigate the described problems by proposing new tools that, by design, are able to both simplify the work of the developers and to greatly reduce hardware requirements. More precisely, three tools are going to be presented, called JavaSPI, iNFAnt and DELTA. These tools are able to face the described issues from orthogonal points of view: for this reason it is theoretically possible to combine all three of them together or to just use one of the three, autonomously.

JavaSPI is a framework, based on Spi2Java,[PS10] allowing to perform the Model-Driven development of security protocols in Java. This means that it enables a developer to build the formal model of a communication protocol like a simple Java application. This model can then be used by existing Model Verifier tools, to automatically build mathematical proofs about security properties of the protocol, or it can be used to semi-automatically build the implementation code of that algorithm. As long as the code generator is trusted, the framework itself can mathematically ensure the soundness relation between the implementation and the model while having built the code through an automated tool further reduces the probability of introducing vulnerabilities in the implementation phase. As its predecessor Spi2Java, this framework is also useful when an implementation of a protocol is already available: in this case the framework can be used to implement mathematically-proven-robust monitoring tools able to ensure that one of the peers of the communication is behaving as expected, thus being able to proactively spot implementation vulnerabilities.[PJ09]

iNFAnt,[CRRS10] instead, is a packet processor based on regular expression processing that allows to exploit the processing power of GPUs to improve the amount of data per second that can be analyzed. Aim of this tool is to reduce the costs of ensuring network security through traffic analysis by enabling to replace expensive special-purpose machines with cheaper general purpose hardware, such as CPUs and GPUs, and still obtain appreciable performance results. This thesis does not cover the entire development of iNFAnt, but it focuses on several optimization techniques that have been added to further improve the performance of this tool. Some of the presented techniques are general to any FSA-based string matching tool, while other are specific for the GPU-accelerated environment.

Finally, the DELTA[ARSBar] framework allows to re-use existing resources to further improve the performance of data analysis tools like iNFAnt and reduce costs: this framework, in fact, allows to develop distributed data analysis tools where a wide range of devices (ideally, all the devices under analysis) can cooperate to analyze their own data. By using this technique, for instance, a data analysis tool like iNFAnt could be distributed across the network so that the devices with a dedicated GPU board can perform part of the string matching task while other devices could pre-process and filter data. If the amount of resources available to be re-used is high enough, this framework could theoretically make completely unnecessary to buy new hardware to analyze the network traffic. Moreover, the framework completely hide the complexities of turning a centralized application into a dynamically distributed one, thus minimizing the level of expertise required to develop these analytics.

1

Introduction

Formal methods, Network Traffic Analysis, Parallel and Distributed Computing. This section provides some brief insights about the research domain on which this thesis is focused by evidentiating our motivations and our contributions. Finally, an outline of the rest of the thesis is provided as brief reference.

Contents

1.1 Domain

Network related security issues are an highly debated topic in literature: several research fields stem from this concept, either focusing on guaranteeing secrecy of information, authenticating the communication actors and so on. All these research fields are born because, actually, developing applications able to use communication channels that malicious attackers could potentially have under control is an extremely complex task: vulnerabilities, in fact, may be hidden at several different abstraction levels, ranging from the communication protocol design to the used ciphering algorithms, and unfortunately it is not actually possible to give a complete guarantee about the absence of any vulnerability.

However, literature provides several tools enabling to enforce security properties with high levels of confidence: two of the possible approaches that could be used to this extent regards defining a mathematical model of the possible attackers in order to formally prove that a particular attacker model cannot violate the desired security properties of the protocol under analysis, or they rely on the fact that, disregarding from the possible presence of vulnerabilities, it is possible to distinguish the patterns of suspicious network traffic produced by a malicious attacker among the traffic produced by legitimate users; for this reason, constantly monitoring the traffic produced in a network may help to identify and block any possible attack.

In this thesis we decided to focus on these two approaches: both techniques are extremely powerful and with a lot of possible application fields, but current implementations have drawbacks that may make them inapplicable in most cases, as it will be detailed in the next sub-sections.

1.1.1 Formal verification of security protocols

This particular research field relies on defining a common mathematical language that could be used to model the communication protocol under analysis, to define the attacker behavior and define set of rules representing the security properties we are interested in. Provided that the attacker model is flexible enough to model a huge range of realistic attacks, and provided that everything is correct, it is possible to connect these models and building mathematical proofs about the presence (or the absence) of the desired security properties.

This technique can obviously build proofs that just holds for the defined attacker model, but typically these models are generic enough to represent an enormous amount of potential attacks. To make an example, the Dolev-Yao model[DY83a] defines an attacker as an entity with complete control over a communication channel that is capable of reading, modifying, erasing and forging messages, by using all the cryptographic functions available to the applications themselves.

These demonstrations are typically not manually performed but, to reduce the probability of mistakes to the minimum, several frameworks have been developed to perform these operations automatically. Moreover, to further reduce the probability of introducing errors when operating, the most sophisticated frameworks are able to automatically derive the mathematical model of a protocol from its own implementation or, otherwise, they are able to automatically generate an implementation of the defined model. The first technique is particularly useful when an implementation of the protocol is already available and we are only interested in proving its properties, while the latter enables to perform the Model Driven Development (MDD) of communication protcols, particularly useful to develop communication protocols by scratch without having to delve in the complexities of writing a reliable implementation of them. In some cases however this technique has also been used to validate existing protocols: even if the implementation of a certain security protocol was already available, in

face, developers built a formally verified implementation of the same one and they used it to monitor the behavior of the real application; in this way the original application, extremely optimized to maximize its performance, was still used, but flanked by its slower automatically generated implementation. When the two applications disagreed on some output it meant that one of the un-modeled vulnerabilities of the original implementation was found. An example of this type of experiments is shown in.[PJ09]

The main flaw of all these techniques, however, regards the extrmely high level of expertise required to properly use all these tools: writing the mathematical model of a communication protocol is not a trivial task, as it usually requires using an exotic formal language, moreover in order to be able to build reliable proofs about the desired security properties it is necessary to know extremely well how the model verifier works, since a huge portion of the work still lays on the shoulders of the users of this software, like for instance the definition of the queries to use to define security properties.

These problems unfortunately limt the applicability of these techniques only in the environments in which a company can afford to pay an additional extremely specialized team of mathematicians and developers able to perform this analysis, while in all the other cases the safest option is still to avoid trying to develop a new communication protocol by scratch but to re-use already existing protocols of proven robustness.

1.1.2 Analyzing network traffic to enforce security

This alternative approach regards acting after that the communication software has been developed by monitoring the usage of the potentially critical devices: the idea is that by analyzing the network data it is theoretically possible to monitor the user behaviors, to detect and eventually block malicious behaviors. A very important advantage of this approach is that it is not necessary to know the original code of the implementations, because the analysis is performed externally from the critical devices.

Depending on the type of attacks that needs to be recognized, several different techniques can be used to enforce security: in some cases it is considered enough just to recognize packet headers, while in other cases it is necessary to reconstruct sessions and analyze data at application level. Being able to perform this last operation opens the so-called “deep packet inspection” research field: it has clear advantages with respect to a simple inspection of the packet headers because it can detect a lot more possible malicious attacks, but at the same time it has drawbacks in terms of costs. Due to the ever increasing bandwidth of network links, in fact, an analyzer has to be able to process huge amounts of data in extremely reduced amounts of time: for this reason several optimizations, both at hardware and software level, are required to be able to keep the pace with the modern network speeds. A common approach is to use powerful special-purpose devices to be able to perform this type of operations, with the clear disadvantage of the extrmely high costs that this choice implies.

1.2 Contribution

The scenario depicted in the previous section decribes an environment in which, in order to be able to guarantee security properties, it is necessary to either have an high level or expertise or an high amount of economical availability. Aim of this thesis is to provide alternative techniques able to contain these problems to a certain extent.

For what concerns the formal verification of security protocols, a new MDD framework is presented, called JavaSPI: this framework enables to define abstract protocol models directly by writing applications in the Java programming language, with the help of an annotation system to define the security property queries: in this way developers can obtain formal proofs about their desired security properties without the need to learn a new modeling language and without having to know too much about the tool used to build the formal proofs. Moreover, thanks to its semi-automated workflow, it guarantees adherence between the formal model and the concrete implementation of the code, thus ensuring that the security properties verified in the model are also true for the implementation.

For what concerns costs related to traffic analysis, instead, two orthogonal techniques are proposed: the first one regards exploiting parallel capabilities of GPUs to obtain good network analysis throughput without the need to buy special-purpose hardware, while the second one regards further distributing a portion of the processing task across the same range of devices under analysis, thus further alleviating the workload that the analyzer has to sustain. These techniques could be applied autonomously or they could also be combined together, enabling to re-use all the devices of a network to analyze its own traffic by also exploiting GPUs for certain specific processing tasks.

Benefits of the described approaches will be shown both in theoretical, general terms and through use cases: famous communication protocols like SSL have been implemented with JavaSPI, while the two data analysis tools called iNFAnt and DELTA have been used to analyze traffic of famous, commercially available packet inspection rule sets and to completely distribute existing data analyzers like MOSAIC.[XSL+13]

1.3 Outline

Here follows a brief description of the thesis structure.

Part I focuses on the Formal Verification research field: in Section 2 some background information about state of the art will be provided, while Section 3 will present the JavaSPI framework. Section 4 explores the theoretical details about JavaSPI by providing formal proofs about the relation between the JavaSPI modeling language and both the automatically generated ProVerif code and concrete Java implemenation codes. Finally, Section 5 shows a practical application example of the JavaSPI framework: in this case the SSL handshake protocol has been used as case study and an interoperable implementation of this protocol has been developed through JavaSPI, by also generating formal proofs about its security properties.

Part II, instead, delves in the traffic monitoring field: after the introductiory background information provided in Section 6, a series of techniques used to accelerate GPU-based packet processing tools like iNFAnt are described in Section 7, while Section 8 provides some of the performance results obtained with the proposed techniques. Then, Section 9 proposes the DELTA framework to further reduce the workload posed on the server by distributing some of the data analysis tasks through the same devices producing the network traffic to be analyzed. In Section 10 a case study will be presented, as the MOSAIC traffic analysis tool will be distributed through through all the devices under analysis, and then it will be compared to its original “centralized” version.

Finally, Part III concludes by summing up the obtained results.

Part I

Building secure applications through Formal Methods

2

Background

The state of the art for what concerns the known techniques to use Formal Methods to develop secure cryptographic communication protocols, with a particular detailed view of the Spi2Java framework, the main starting point used to develop JavaSPI

Contents

2.1 Formal methods and security protocols

Security protocols are communication protocols that aim to reach some goals despite the hostile activity of attackers that interfere with the protocol (e.g. by having access to the public channels used by protocol actors). Typical goals are concealing information to unauthorized parties or giving one actor assurance about the identity of another actor with which it is communicating. The typical means used for this purpose is cryptography.

Security protocols are generally used to protect something valuable. This is why high assurance about their correctness is highly desirable. Unfortunately, despite their simplicity, security protocols are quite difficult to get right.

The main difficulties, experienced even by security experts, are not just related to the strength of the cryptographic algorithms employed (even if these problems must be faced too); when designing a novel security protocol it is necessary to take into consideration all possible behaviors of hypothetical attackers, including violations of the protocol rules, and any possible forgery of messages. The number of these behaviors is typically unbounded or at least huge, because an attacker can forge and inject a new message at each protocol step in a number of ways that is typically unbounded. This fact adds extra complexity to the already complex concurrent interactions that a communication protocol must normally manage. Thus, despite the existence of best practices and recommendations,[AN96] the manual design of a novel security protocol remains a very error-prone and challenging task. The difficulty of defining security protocols right is witnessed by stories like the one of the Needham-Schroeder public-key protocol,[NS78] which was believed secure for 17 years before Lowe discovered it was affected by a flaw;[Low95] another witness is the recent discovery of a logical flaw[i] in the renegotiation feature of the widely used TLS protocol,[DR08] 13 years after the first version of the protocol was published (under the SSL 3.0 name).

Due to the inherent complexity, developing security protocols right demands rigorous, mathematically based methods for reasoning about their correctness. It is significant, for example, that the above mentioned flaw affecting the Needham-Schroeder public-key protocol could be found by applying formal methods.[Low95]

The rigorous methods that have been developed so far are mainly based on using abstract models of the security protocols under analysis. Depending on the level of abstraction of these models two different lines of research can be delineated: at the highest level there is the symbolic approach, originated from the seminal paper by Dolev and Yao,[DY83b] that considers cryptographic functions as ideal: it is very easy to build formal proofs about security properties with these techniques, but at the same time the model is so abstract that the generated security property proofs may be extremely complex and difficult to relate to the real world; at a lower level, instead, we can find the Computational approach: originated from the papers by Goldwasser and Micali[GM84] and by Yao,[Yao82] this technique involves complexity and probability theories to be able to also consider weaknesses of cryptographic algorithms; thanks to this fact this method gives more realistic security assurances, at the expense of increased difficulty in proof automation.

Both symbolic and computational approaches provide rigorous proofs based on abstract models, albeit at different levels of abstraction. In spite of this, a large gap still exists between these models and a real-world protocol implementation and its execution. This gap may be responsible for final unsatisfactory security levels, even when correctness proofs have been developed from a model of the protocol. One important component of this gap is the usually big difference between an abstract protocol model on which proofs are developed and the real code that implements the protocol, written in programming languages. For example, the real control flow and data types of a protocol

[i] http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-3555

implementation are generally more complex than the ones of the abstract models. Moreover, when deriving an implementation from a model or specification, programmers may introduce logical and coding errors, some of these errors may not be detected by testing and may make the behavior of the implementation not corresponding to the model or specification. In practice, widely spread implementations of security protocols, such as OpenSSL and OpenSSH, receive several security patches per year, due to low-level implementation bugs. To make an example, in OpenSSL an error condition returned by a cryptographic function was incorrectly interpreted by the function caller, making the application accept corrupted data;[ii] such a fault cannot be found if a formal model that has no relation with the implementation code is analyzed, because the semantics of the model itself defines the (correct) interpretation of the results of cryptographic functions, and the way the code handles return values is neglected.

Additionally, each programming language has its own mechanisms for accessing data and its own libraries for performing basic operations. Of course, these details cannot be considered by language-agnostic abstract models like the ones that are usually analyzed in a rigorous way, and may be responsible for program bugs that affect security.

On the basis of such considerations, in recent years some researchers have started working towards methods that reduce the gap between models and implementations, bringing formal security proofs closer to real protocol implementations.[APS14] These techniques are based on tools that helps developers to build formal models starting from the source code of a communication protocol or vice versa. Depending on the tools, this operation can be completely automated or it can still require the help of the developers. One of the most complete frameworks performing this type of operations is Spi2Java.[PSD04]

2.2 Spi2Java

One of the most classical examples of framework to perform model driven development of security protocols by generating symbolic proofs about its security properties is Spi2Java[iii][PSD04, PS07] : this framework models protocol in spi calculus, a formal process algebraic language.

Each spi calculus specification is a system of independent processes, executed in parallel and exchanging messages on shared communication channels. Each process represents the execution of a single run (or session) of the protocol by a single actor. Accordingly, a process is typically described as a (sequential) program, made of expressions such as message transmission or reception and application of cryptographic functions and checks on messages.

Messages are represented symbolically as terms of an algebra, and cryptographic functions as algebraic operators on these terms. Such operators have the properties that the corresponding cryptographic functions should ideally fulfill. For example, as $H(x)$ represents the hashing of x, $H(a)$ and $H(b)$ are always different for different a and b, and there is no operator that takes $H(x)$ and returns x. Terms are untyped, in order for the model to be able to represent possible attacks based on type confusion.

With this language it is possible to write an abstract model of a protocol to automatically analyze it and formally verify that there are no possible attacks on the protocol under the modeling assumptions made. Of course, this requires the protocol expected goals to be formally specified too. The analysis can be done, for example, by the automatic theorem prover ProVerif,[Bla09a] that can work on spi calculus.

[ii] `http://www.openssl.org/news/secadv_20090107.txt`
[iii] http://www.spi2java.com

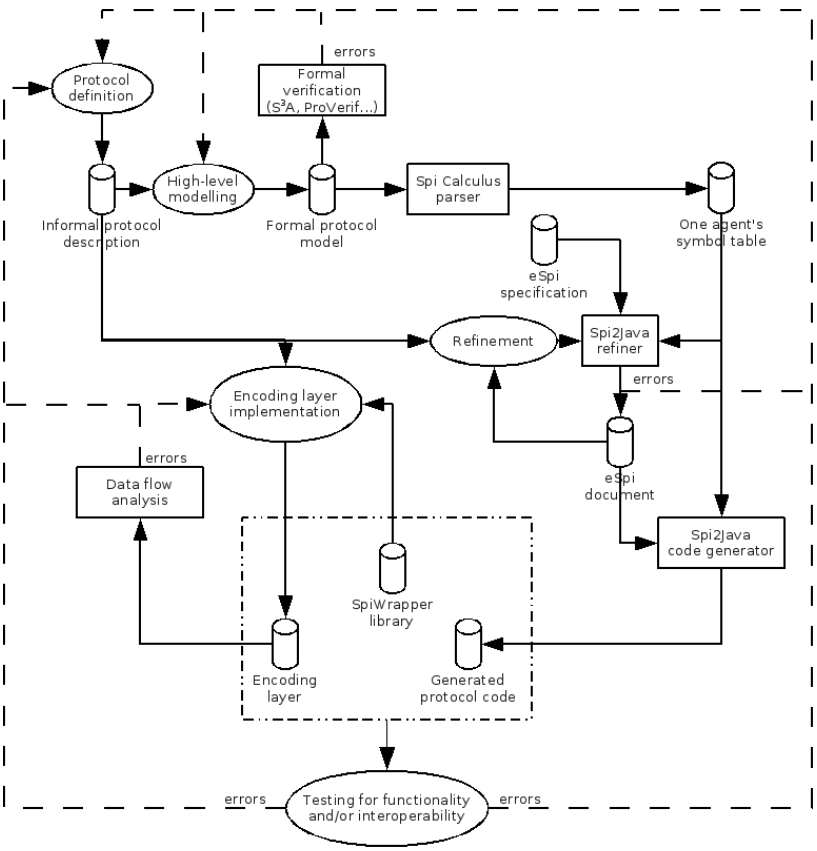

Figure 2.1: Spi2Java methodology workflow

Once the abstract model has been successfully analyzed, and it has been shown that it is free from logical flaws, a Java implementation can be derived for each protocol role.

During this refinement step, the abstract model must be enriched with all the missing protocol aspects that are needed in order to get a concrete and interoperable Java implementation: (i) concrete Java implementations of cryptographic algorithms with their actual parameters; (ii) Java types to be used for terms; and (iii) concrete binary representations of messages and corresponding Java implementations of marshaling functions.

The Spi2Java framework also requires the user to manually edit and keep in sync the model and an intermediate XML file containing refinement information, which is error prone and time consuming.

The full operating workflow of the Spi2Java framework is proposed in Figure 2.1.

In conclusion, this framework is extremely powerful as it allows to generate new interoperable implementations of security protocols on which the most critical part of the code is proven to be free from vulnerabilities and implementation issues. Moreover, adherence between the abstract model and the implementation is mathematically proven by a soundness theorem.[PS10, PS12]

However, using this tool is not trivial unless the developer has an high level of expertise in this field: it requires, in fact, to know how to develop a model with the spi calculus syntax, how to verify security properties by writing queries in ProVerif and how to interpret the traces ProVerif generates when it finds a vulnerability. Not having a complete control over this knowledge could be very dangerous because there is the risk to generate proofs that are pointless since they have been generated from queries not representing real security properties. For these reasons we based our new framework, JavaSPI, on Spi2Java, as it represents one of the most complete MDD frameworks that leads the developers from the abstract model to the implementation, and then we took care of all the complex

aspects of Spi2Java in order to build tools that are easier to use, less error prone and as powerful as the original ones.

3

The JavaSPI Architecture

Delving into details of the JavaSPI framework by describing its expected operating workflow, its architecture and its potential: why using the JavaSPI framework represents an improvments to the state of the art and how it allows to obtain the same results of more complex frameworks by reducing the expertise required from its users.

Contents

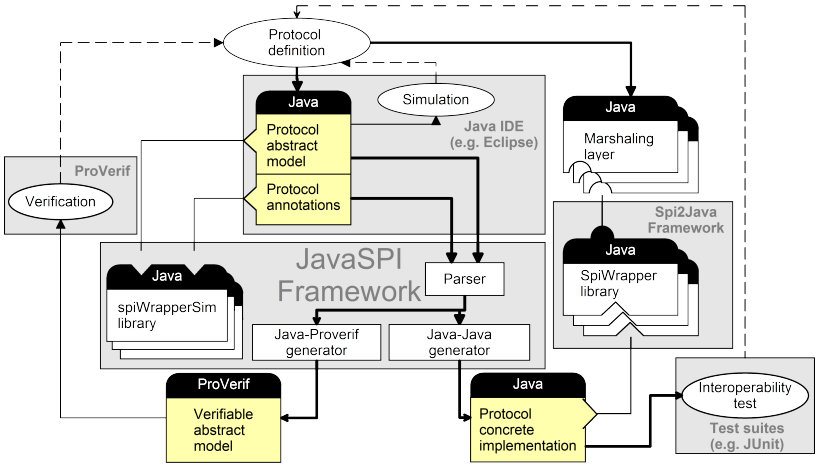

Figure 3.1: The complete workflow provided by JavaSPI

3.1 Working principles of JavaSPI

JavaSPI has been developed as a set of tools and utilities enabling the user to model a cryptographic protocol by following the workflow shown in Figure 3.1: basically, the user is intended to develop abstract models in the form of typical Java applications, but using a specific library that is part of the JavaSPI framework, named *SpiWrapperSim*, that contains a set of basic data types along with networking and cryptographic primitives.

The logical execution of the protocol can be simulated by simply compiling and debugging the abstract code. The protocol security properties can then be formally verified by using the JavaSPI *Java-ProVerif* converter that produces an output compatible with the ProVerif tool.

Once a model has been properly designed, it can be refined by adding implementation information by means of Java annotations, as defined in the SpiWrapperSim library. From the annotated Java model a concrete implementation of the protocol can be generated by using the JavaSPI *Java-Java* converter.

The entire JavaSPI framework described here has been completely developed from scratch: still, some architectural choices have been made to allow re-use of parts of the Spi2Java framework.

3.2 Writing the abstract model

The JavaSPI framework includes a Java library, called SpiWrapperSim, that can be used to write abstract security protocol models as Java applications and to simulate them.

Models that can be expressed in this way are instances of the class of models that can be described by the input language of ProVerif. Based on this, the framework provides the *Java-ProVerif* tool that transforms a Java model into the corresponding ProVerif model, compatible with ProVerif. Note that differently from other papers like,[BFGT08] here the ProVerif model is not *extracted* from the Java code, rather the model, expressed in the Java syntax, is translated into the ProVerif syntax. A Java model differs from the final Java implementation because it is as abstract as the ProVerif model.

Java abstract model

```
1 Message m = new Identifier("Secret message");
2 Nonce n = new Nonce();
3 SharedKey s = new SharedKey(n);
4 SharedKeyCiphered<Message> mk =
   new SharedKeyCiphered<Message>(m,s);
```

Java concrete implementation

```
1 Message m =
     new IdentifierSR("Secret message");
2 Nonce n = new NonceSR("8");
3 SharedKey s =
     new SharedKeySR(n, "DES", "64");
4 SharedKeyCiphered mk =
   new SharedKeyCipheredSR(m, s, "DES",
         "1234567801g=", "CBC",
         "PKCS5Padding", "SunJCE");
```

ProVerif model

```
1 new m1;
2 new n2;
3 let s4 = SharedKey(n2) in
4 let mk6 = SymEncrypt(s4, m1) in
```

Figure 3.2: An example of how four lines of the abstract model are converted into the corresponding concrete implementation and ProVerif syntax.

Moreover, the Java model can also be executed like any regular Java application. Its execution in fact simulates the underlying model that it describes, thus giving the user the possibility to debug the abstract model. In this execution messages are represented symbolically, and input/output operations are implemented by exchanging symbolic expressions over in-memory channels behaving according to the classical spi calculus semantics.

In order to get a Java program that models a protocol in this way, the user must use Java according to a particular programming pattern. Only the SpiWrapperSim library can be used for cryptographic and input/output operations, and some restrictions on the Java language constructs that can be used for the description of each process apply. These restrictions, documented in the library JavaDoc, naturally lead the user to develop models in the right way.

A protocol role (a "process") is represented by a class that inherits from the library class *spiProcess*. In this way, the common code needed for simulation that surrounds the protocol algorithm is hidden inside the superclass. Moreover, objects derived from *spiProcess* are allowed to use some protected methods that enable common operations, like the parallel instantiation of sub-processes.

The class that inherits from *spiProcess* must define the *doRun()* method, that is the abstract description of the protocol role.

Any message, complex at will, can be represented by an immutable object belonging to a class that inherits from the *Packet* library class. The fields of this class are the fields of the message. The class must be made immutable by declaring all fields as final. This is necessary as, in spi calculus, each variable can be bound only once. Using mutable Java objects would be possible but it would then entail more complex relationships between the Java code and the underlying model.

The only class types the user is allowed to instantiate are the ones provided by the SpiWrapperSim library, plus the ones used as arguments of methods of such classes (e.g. *String*). The primitive type *int* is also admitted, but only for loop control flow, with the constraint that each loop must be bounded and the bound must be known at compile time.

Conditional statements are possible only with equality tests (via the *equals()* method) and with tests on the return values of certain operations of the library.

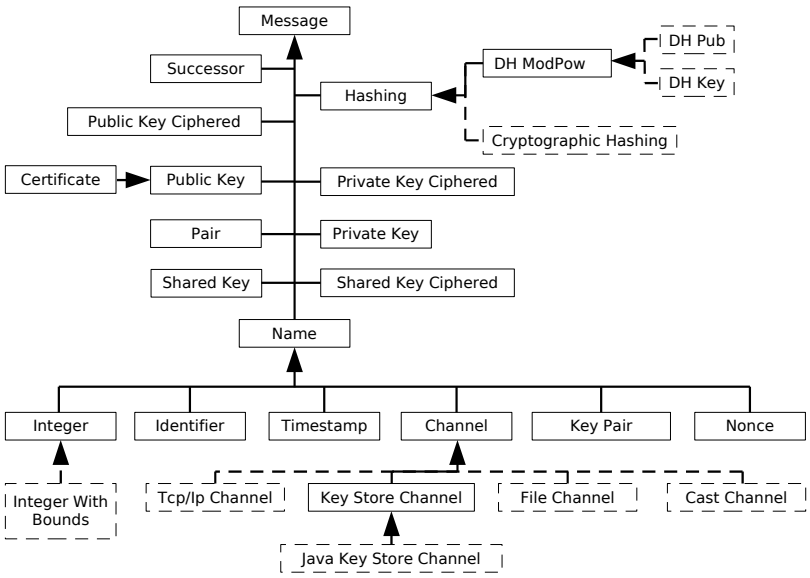

Figure 3.3: The complete data types used by JavaSPI and Spi2Java

SpiWrapperSim is very similar to the SpiWrapper library that provides the implementations of the spi calculus cryptographic and communication operations in the Spi2Java framework. This is a precise architectural choice that greatly facilitates the last development step, i.e. the refinement of the abstract model into a concrete implementation. Indeed, the implementation code is based on the SpiWrapper library.

As it is possible to notice in Figure 3.2, thanks to this choice even the syntax used in the two codes is very similar; the main difference is just that the abstract model lacks many implementation details, like the encryption algorithms of each cryptographic function call, or the marshaling functions (whose implementation is included in the "SR" suffixed classes in the example shown).

Figure 3.3 shows the full set of data type elements defined both by the SpiWrapperSim and by the SpiWrapper libraries. As SpiWrapperSim aims to be a simplified library, certain particular elements (expressed with dashed lines) have been omitted in SpiWrapperSim while they still exist in SpiWrapper.

From the functional point of view, applications developed by using the SpiWrapperSim library does not really have cryptographic functionalities, especially because the library itself does not allow to specifiy parameters that would be crucial to perform them: the SpiWrapperSim objects, instead, perform a symbolical simulation of the cryptographic functions by referring to the ideal behavior each cryptographic function is expected to have. To name an example, the *Hashing* object does not perform any real mathematical function but it just internally stores a reference to the object on which the hash has to be calculated. The method *Hashing.equals(o)*, then, works by comparing the objects the two Hashing instances are encapsulating.

Finally, the SpiWrapperSim library also provides a set of annotations that can be used during refinement to assign, for each object, its implementation details. As annotations do not affect the simulation phase, they can be specified later on, just before generating the concrete implementation.

By using this technique the implementation details and the code both reside on the same file: this means that JavaSPI is not affected by the sync problems described previously for Spi2Java. Moreover, each annotation has a scope and a default value, so that it is not necessary to specify each implementation detail for each object used in the code, but it is possible to specify just the implementation details that differ from the default values (or it is even possible to just tune the default values).

Table 3.1: A significant portion of the conversion mapping between the Java model and ProVerif model.

Statement	Java	ProVerif
Fresh	**Type** $a = new$ **Type** ();	$new\ a;$
Assign	**Type** $a = b;$	$let\ a = b\ in$
Hashing	$Hashing\ a =$ $new\ Hashing(b);$	$let\ a =$ $H(b)\ in$
Send	$cAB.send(a);$	$out(cAB, a);$
Receive	**Type** $a =$ $cAB.receive($ **Type**$.class);$	$in(cAB, a);$
SharedKey	$SharedKey\ key =$ $new\ SharedKey(a);$	$let\ key =$ $SharedKey(a)\ in$
Encrypt	$SharedKeyCiphered$ $<$ **Type** $> a = new$ $SharedKeyCiphered$ $<$ **Type** $> (b, key);$	$let\ a =$ $SymEncrypt$ $(key, b)\ in$
Decrypt	**Type** $a =$ $b.decrypt(key);$	$let\ a =$ $SymDecrypt$ $(key, b)\ in$
Error handled Decipher	$ResultContainer$ $<$**Type** $> c =$ $a.decrypt\_w(key);$ $if(c.isValid()\{$ **Type** $b =$ $c.getResult();$ $...\}else\{...\}$	$let\ b =$ $SymDecrypt$ $(key, a)\ in\ ($ $...$ $)else($ $...$ $)$
Packet Comp.	**PacketType** $m = new$ **PacketType**$(a, b, ...)$	$let\ m =$ $(a, b, ...)\ in$
Packet Split	**Type** $a =$ $b.get$**Field**$();$	$let\ a =$ b_get**Field** in $(*)$
Match case	$if(a.equals(b))\{$ $...\}else\{...\}$	$if\ a = b\ then($ $...)else(...)$
Start	$SpiProcess\ a =$ $new\ Client(c, d, ...);$ $SpiProcess\ b =$ $new\ Server(e, f, ...);$ $start(a, b);$	$(Client(c, d, ...)\|$ $Server(e, f, ...))$

Type stands for any class name, **PacketType** stands for any user-defined Packet class name, **Field** stands for any field name in a Packet class, while **a**,... **f** and **key** stand for variable names.
(*) Variable *b_getField* is created in ProVerif code during a Packet splitting phase which is automatically generated after any Decrypt or Receive statement that produces a Packet object.

By following the intended workflow, the Java model can be converted to a ProVerif compatible model, or a concrete Java implementation can be derived from the Java model. The next two subsections will cover these two cases.

3.3 Building formal security proofs

The mapping from Java to ProVerif syntax is based on simple rules, developed in this work along with the corresponding converter, that are informally exemplified in Table 3.1. Each Java statement that may occur in a *doRun* method is mapped to a corresponding ProVerif equivalent piece of code. For simplicity, the figure does not consider the addition of the numeric suffix in ProVerif, needed in order to disambiguate variable names, as shown in Figure 3.2.

Conversion of loops requires special handling. ProVerif does not support unbounded loops natively, but they can be easily encoded as recursive processes. However, ProVerif often experiences termination problems when loops encoded as recursive processes are used. Because of this limitation of the verification engine, the restriction of having only bounded loops was introduced in the Java modeling language, so that the conversion tool can perform loop unrolling in order to eliminate loops.

The fields of a Java Packet object are translated into nested pairs. In order to facilitate code translation and readability, a new variable is introduced in ProVerif for each field. For example, let us consider a class called MyPacket with three fields called a, b and c, all of type Nonce. The Java code

```
MyPacket p = channel.receive(MyPacket.class);
Nonce a = p.getA();
Nonce b = p.getB();
Nonce c = p.getC();
```

that receives a message of type MyPacket and extracts its three fields is converted into the following ProVerif code:

```
in(channel1, p2);
  (* Packet expansion *)
  let p2_getA3 = GetLeft(p2);
  let tmp4 = GetRight(p2);
  let p2_getB5 = GetLeft(tmp4);
  let p2_getC6 = GetRight(tmp4);
(* Variable assignment *)
let a7 = p2_getA3;
let b8 = p2_getB5;
let c9 = p2_getC6;
```

By using this technique the converter is forced to write, in ProVerif, more code lines than with the Java syntax, but this disadvantage is overcome by the fact that this technique totally hides to ProVerif the additional complexity that custom packet types could cause, thus avoiding the risk to generate a model that is too complex to be used to build formal proofs, often referred as diverging code.

There is also another particular characteristic of ProVerif which actually needs to be taken in consideration: this syntax, in fact, does not allow to write any line of code after an if/else statement. This poses some limits to the Java-ProVerif conversion, as it generates some situations in which a simple rule-based mapping is not feasible.

The initial solution to this problem has been to forbid the users to write code after an if/else statement. This, however, limits the expressiveness freedom a Java developer usually have: for this reason it has been studied a new technique based on a pre-conversion step, internally executed before the real Java-ProVerif conversion, that cuts the code after each if/else statement to paste it in the end of each conditional branch. This operation, again, generates a ProVerif file that can potentially be much more complex than the Java model, but this can be considered an acceptable tradeoff, as in this way it is not necessary to limit the developer expressiveness power. Moreover, ProVerif file is not meant to be read by any developers, it just needs to be used by the corresponding model checker.

Translating plain Java models into ProVerif is not enough to enable automatic verification of security properties. Indeed, the ProVerif toolchain still needs to have two types of information:

- It is needed to shape the attacker knowledge base
- ProVerif will need to know which security properties have to be checked.

The first step is relatively easy to perform: usually, practically any initialization variable needs to be considered as public, along with the communication channels. However, sometimes it could be useful to express constraints in a more complex way, as example when a particular channel can be considered safe or something similar.

For this reason, the attacker knowledge base is expressed by using a single annotation, called *@pVarDef(PRIVATE|PUBLIC)*. This annotation can be applied to a single variable or to an entire block of code: in this last case every variable declared inside the code block inherits the *pVarDef* property that is closer to him, with the *PRIVATE* property as default.

With these simple rules it is possible to express very complex knowledge bases with a very small effort: in fact, in a simple protocol, the files that model the actor behaviours doesn't need these annotations. The *pVarDef* annotation is just written on the instancer process, by defaulting its variables as *PUBLIC*. Changing this behaviour just implies adding few annotations on some variables in the instancer, when these variables must be considered *PRIVATE*.

Please note that the *pVarDef* annotation has a direct influence on how the ProVerif code is generated: in fact every *PUBLIC* variable is declared as a free or constant term (whether if the variable is a channel or any other data type), that are particular elements globally available througout the entire protocol code. As this behaviour is not logically the same of the Java model, a particular variable renaming technique has been applied in order to avoid name conflicts and other disaccording behaviours.

For what concerns the listing of security properties in the Java model, the JavaSPI library provides a specific annotation set. These annotations are then processed during conversion to ProVerif and translated into corresponding queries in the output ProVerif code.

A variable can be marked as *@Secret* in order to specify that ProVerif should verify its secrecy, in this way:

```
@Secret Nonce PLx = new Nonce();
```

The corresponding ProVerif generated code will look like this:

```
(* Secrecy queries *)
query attacker:PLx21.
```

If the *@Secret* term is a compound term, or anyway a term that needs to be derived from another one the syntax is slightly more complicated: in fact, as ProVerif cannot directly verify the secrecy of variables, the ProVerif query that will be generated will regard the entire composition of the term, along with queries about the secrecy of any base data type involved in the composition. For this reason, during the ProVerif verification some false alerts may happen, because maybe a complex secret term contains in its composition some publicly available terms.

In this case it is completely a developer task to recognize these events and safely ignore them when it happens. In fact actually the interpretation of the ProVerif verification output is still under development, and hopefully these flaws will be partially (or completely) automated in the next versions of the framework.

In order to verify authentication properties, instead, it is possible to use correspondence assertions on the order of events. In JavaSPI, a process can rise an event by calling the *event(String name, Message... data)* method provided by the *SpiProcess* class, where *name* specifies the name of the event, and *data* the data associated to that event. This method has no effect in the code, but it is translated to a corresponding event in ProVerif. When the event sets are defined it is possible to use them to write some interrogations: the reachability of every event, as example, is automatically queried, while in order to check other more complex properties a set of annotations is provided: as example, the correspondence between events, such as "if $event(n1, x)$ happened, then $event(n2, x)$ must have

happened before" can be specified by the *@PEvinj* annotation, associated with the instantiation process class:

```
@PEvinj({"n1", "n2"})
public class Master extends SpiProcess ...
```

This technique can be used to write more advanced queries, by extending the number of events in a *PEvinj* clause to three or more, or by combining multiple *PEvinj* annotations by using another annotation, called *PInjList*, like in this example:

```
@PInjList({
  @PEvinj({"n1","n2","n3"}),
  @PEvinj({"m1", "m2"})
})
...
```

With this set of techniques an user should be easily able to express the main part of basic ProVerif queries. There is still the possibility, however, that the user needs to write a more complex interrogation, not expressible with just these annotations. For this reason a particular annotation has been provided to enable the user to directly write a custom query with the ProVerif syntax. This, however, is an advanced feature that can just be used by experienced developers which actually knows the ProVerif query syntax: for this reason, it is a feature of few interest for the purposes of this thesis, and it will not be discussed in more detail.

3.4 Generating the implementation

The last development stage is the automatic generation of the protocol implementation code from the model. As SpiWrapperSim is similar to the library used for the concrete implementation, there is a strict correspondence between the abstract code (the model) and the concrete code (the implementation). The implementation aspects that are missing in the abstract model can all be specified by means of annotations.

One of such aspects is the choice of the marshaling functions to be used for each object. A default marshaling mechanism based on Java serialization is provided by a library called *spiWrapperSR*, that extends spiWrapper. The user can provide custom implementations of the marshaling functions. This is a key factor enabling development of interoperable implementations of standard protocols, where the specific marshaling functions to be used are specified by the protocol standard.

Another key feature of JavaSPI enabling interoperability is the ability of resolving Java annotations values either statically at compile time, or dynamically at run time. For example, this enables implementations of protocols featuring algorithm negotiation.

Finally, JavaSPI allows to specify how the various constants of the protocol has to be initialized. This is not a trivial task, as sometimes different actors of a procotol may need different constants. For this reason it is necessary to specify, for each actor, a piece of code that initializes every parameter before calling the protocol method in the proper way.

This problem has been solved by enabling each class that inherits *SpiProcess* to override a particular method, called *doInit*. This method is not considered in the simulation and neither in the proVerif verification, but it just contains a custom initialization code that will be integrally replicated in the concrete Java implementation. By using this technique it is possible to minimize the effort required to make the concrete implementation to work. The only "post-generation" modifications

that eventually will still be needed will just regard the integration of the generated code with the rest of the application.

4

Formal definition of the JavaSPI framework

Mathematical definition of the JavaSPI models, the ProVerif language, the Java syntax subset used in the concrete algorithm implementations and mathematical proofs about the soundness relation provided by the JavaSPI framework between the ProVerif model and the concrete Java implementation.

Contents

Aim of this chapter is to prove that a Soundness relation exists between the abstract protocol models generated by JavaSPI and their corresponding concrete Java implementations. This means mathematically proving that the behavior of the ProVerif model, used to prove/disprove robustness of the algorithm to security properties, maps at least all the possible behaviors the concrete implementation code may show. As implication of this proof, if a certain execution trace of the real code leads to a security property violation, this trace is necessarily present also in the abstract model, and it can be found during the model verification stage. Reverting this definition implies that any security property that has been proven to the abstract model can be extended to the concrete implementation as well.

Before delineating and proving the soundness theorem, however, it is necessary to perform a series of preliminary steps, like providing a formal definition of the Java syntax sub-set used by the JavaSPI model and of the formal languages used for verification, expressed using the same kind of "evolution rules", representing a symbolical definition of the behavior the code has while it is being executed at runtime. Section 4.1 defines these evolution rules for all the three used languages, wile Section 4.2 formally defines the translation rules used by JavaSPI to perform conversions between the languages. Finally, once all these building blocks have been formally defined, Section 4.3 introduces and builds the proofs of the Soundness theorem.

4.1 Formalizing the languages

Literature provides a particular formal model, called "Middleweight Java" (MJ) that consider a subset of the Java syntax whose evolution rules have been formally defined in.[BPP03] Both the JavaSPI abstract models and concrete Java implementations generated by the JavaSPI framework respect limits imposed by MJ syntax and, for this reason, it is possible to exploit MJ evolution rules to formally analyze both the pieces of code.

MJ model is based on the concept of "execution state", representing the snapshot of an algorithm in a precise instant of its processing. These states are composed of a sorted set containing the code statements that still need to be processed and on other structures defining the state of the memory the application has allocated. More details can be found in,[BPP03] as they are not reported here since they are outside the scope of this thesis.

Even if MJ evolution rules can be used to model JavaSPI pieces of code, in fact, they are overcomplicated for our needs, especially because they are very distant from the ProVerif evolution rules on which we want to define a relation. For this reason "compound" evolution steps $A \xrightarrow{*} B$ are defined: these steps relate execution states A and B by stating that, by following a certain amount of MJ evolution rules, it is possible to evolve a JavaSPI model or a concrete Java application from state A to state B. By using compound transition functions in place of MJ evolution rules it is possible to hide complexities of the model: for instance, a single compound evolution rule can be used to model an entire method call of a well known library object.

The entire execution model of JavaSPI and the subset of Java used to generate concrete implementations can be expressed with a fixed set of simple compound evolution rules: this "new" execution model is way simpler than MJ while, at the same time, it does not need to be further validated since it relies on series of well known and already validated MJ evolution steps.

An additional simplification regards memory representation. JavaSPI syntax, in fact, limits the life-cycle of a variable to be as simple as possible: when a variable is declared and initialized it becomes read-only and it remains visible until the end of the protocol, thus totally avoiding problematics like "out-of-scope" variable statuses. For this reason the complex memory representation presented in[BPP03] has been totally replaced with a much simpler partial function σ, mapping each variable

name to its value. Adherence to MJ memory representation can be easily proven by exploiting the restrictions imposed on the JavaSPI syntax as presented in Section 4.1.1.

Finally, notation used to represent cryptographically manipulated data in σ uses the syntax defined in.[PS10] For instance, to represent the fact that σ contains a variable x representing a message M encrypted with a symmetric algorithm and a shared-key k, the following notation will be used: $\{x \rightarrow \{M\}_k\} \in \sigma$.

4.1.1 JavaSPI evolution rules

JavaSPI does not allow to use the full Java syntax but it poses several limits. These are the following:

- Every variable must be declared as **final**: thus, once initialized it cannot be re-assigned
- Anonymous variables are not allowed
- The only allowed data types are the ones defined in the **spiWrapperSim** data library: any other data type is forbidden in the Protocol section of the code.
- Any statement that alters the execution flow, like an **if..else** statement or a call to another user-defined method cannot be followed by any other statement.

Please note JavaSPI framework provides a pre-processing engine able to process "syntactic sugar" patterns: it recognizes particular coding patterns able to overcome these limits and it converts the code to a form that does not violate these limits anymore. As this pre-processor does not change the execution model, it has not been considered in this section.

The last presented limit allows to perform the aforecited memory simplification. It implies, in fact, that in any execution trace there will never be a temporarily "out of scope" variable able to come back "in scope" in a second time: for this reason the partial function σ is enough to fully represent the state of a JavaSPI application in any point of any execution trace.

From a general point of view is assumed that a JavaSPI application is composed on a group of parallel threads, representing each one an actor of the communication. The state of a thread is represented by combining a sorted set of JavaSPI statements that have to be executed, P, and the partial function σ, representing the memory of the thread.

It is assumed that, after a certain amount steps (needed to "initialize" the system), the state of a JavaSPI application can be represented as a group of sets $G = \{P_0\sigma_0, P_1\sigma_1, ..., P_n\sigma_n\}$ in which each $P_i\sigma_i$ term represent the state of one of the instantiated threads. Each evolution "step" consists in processing the first statement of one of the actors according to the evolution rules presented later in this section. As we are dealing with parallel processes, each actor may evolve independently from the other ones, thus there are no general rules regarding the order used to choose the actors that must evolve first, apart in the case of a channel data transfer: in these cases, in fact, the actor reading from a channel has to wait until someone sends something in the same channel.

In parallel to G, the concept of "environment" is defined, representing all the data that passes through public channels. The semantics of JavaSPI has been defined by means of a labeled transition system (LTS), similar to the LTS defined in.[PS10] This means that, depending on the environment interactions, two types of labeled evolution rules can be distinguished: τ transitions can be used to evolve a $P\sigma$ thread state anytime ($P\sigma \overset{\tau*}{\rightarrow} P'\sigma'$), without interacting with the environment; on the other hand, $Q\sigma \overset{m!N}{\rightarrow} Q'\sigma'$ and $Q\sigma \overset{m?N}{\rightarrow} Q'\sigma'$ transitions are used when the instruction respectively sends and receives the data N through the channel m. This type of evolution rules, called λ evolution rules, are always defined in pairs so that receiving data from a channel is only possible if someone else, at the same time, sends data to the same channel.

$`RC < Type > v = km.decrypt\_w(key);' \ P,$	$\xrightarrow{\tau*}$	$P,$
$\{km \to \{M\}_K\} \cup \{key \to K\} \in \sigma$		$\{v \to (TRUE, M)\} \cup \sigma$
$`RC < Type > v = km.decrypt\_w(key);' \ P,$	$\xrightarrow{\tau*}$	$P,$
$\{km \to \{M\}_K\} \cup \{key \to K'\} \in \sigma$		$\{v \to (FALSE, N)\} \cup \sigma$
$`if(v.isValid())\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau*}$	P, σ
$\{v \to (TRUE, M)\} \in \sigma$		
$`if(v.isValid())\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau*}$	Q, σ
$\{v \to (FALSE, N)\} \in \sigma$		
$`Typet = v.getValue();' \ P,$	$\xrightarrow{\tau*}$	$P,$
$\{v \to (TRUE, M)\} \in \sigma$		$\{t \to M\} \cup \sigma$
$`if(a.equals(b)\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau*}$	P, σ
$\{a \to M\} \cup \{b \to M\} \in \sigma$		
$`if(a.equals(b)\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau*}$	Q, σ
$\{a \to M\} \cup \{b \to N\} \in \sigma$		

Table 4.1: Some of the JavaSPI τ evolution rules

$`c.send(o);' \ P,$	$\xrightarrow{c!M}$	P, σ
$\{o \to M\} \in \sigma$		
$`final \ Type \ t = c.receive(Type.class);' \ P,$	$\xrightarrow{c?M}$	$P,$
σ		$\{t \to M\} \cup \sigma$

Table 4.2: JavaSPI λ evolution rules

Table 4.1 presents a subset of the $\xrightarrow{\tau*}$ transition rules, while the λ evolution rules are listed in Table 4.2. For a full list of evolution rules, please refer to Appendix A.1. As P is a sorted set, in these tables the space has been used as conventional concatenation symbol, while strings delimited by single quotes (‘ ’) represent one or more statements involved in that evolution rule. In some statements a set of tokens have been used to represent particular groups of objects: for instance, the *Type* token represent a name of one of the data types of the **spiWrapperSim** library (the full list of data types can be found in[APSP11]), while *RC* is just a placeholder to shorten the longer *ResourceContainer* type name.

In these tables it is assumed that the application is “well formed”, in the sense that it must already satisfy preconditions needed to make it a compilable and runnable Java statement. Preconditions presented in the following tables are just needed as they semantically affect the code, while thanks to the “well-formedness” statement, any other precondition regarding, for instance, type safety bounds does not need to be reported in these rules.

4.1.2 The Java Implementation

As concrete Java code is semi-automatically generated from JavaSPI models, semantical behavior of these implementations is nearly identical to the behavior of initial JavaSPI models: for this reason, rather than formalizing the entire Java semantics, we just focus on the restricted syntax allowed by the code generator. All the limits and simplifications already presented for JavaSPI are still valid in this case, and thus evolution rules are very similar to the previously defined ones. The only difference regards the presence of additional implementation details in any method call. These details are automatically generated according to some annotations the user can put in the JavaSPI code.

These additional implementation details impacts to the evolution rules: a new failure scenario, in fact, has been defined to consider the case in which any cryptography operation fails due to the presence of wrong, non-compatible parameters.

$`RC < TypeCC > v = km.decrypt\_w(key, par);' \ P,$	$\xrightarrow{\tau^*}$	$P,$
$\{km \to \{M\}_K{}^{par}\} \cup \{key \to K\} \in \sigma$		$\{v \to (TRUE, M)\} \cup \sigma$
$`RC < TypeCC > v = km.decrypt\_w(key, par);' \ P,$	$\xrightarrow{\tau^*}$	$P,$
$\{km \to \{M\}_K{}^{par'}\} \cup \{key \to K\} \in \sigma$		$\{v \to (FALSE, N)\} \cup \sigma$
$`RC < Type > v = km.decrypt\_w(key, par);' \ P,$	$\xrightarrow{\tau^*}$	$P,$
$\{km \to \{M\}_K{}^{par}\} \cup \{key \to K'\} \in \sigma$		$\{v \to (FALSE, N)\} \cup \sigma$
$`if(v.isValid())\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau^*}$	P, σ
$\{v \to (TRUE, M)\} \in \sigma$		
$`if(v.isValid())\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau^*}$	Q, σ
$\{v \to (FALSE, N)\} \in \sigma$		
$`Typet = v.getValue();' \ P,$	$\xrightarrow{\tau^*}$	$P,$
$\{v \to (TRUE, M)\} \in \sigma$		$\{t \to M\} \cup \sigma$
$`if(a.equals(b)\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau^*}$	P, σ
$\{a \to M^{par}\} \cup \{b \to M^{par}\} \in \sigma$		
$`if(a.equals(b)\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau^*}$	Q, σ
$\{a \to M^{par}\} \cup \{b \to M^{par'}\} \in \sigma$		
$`if(a.equals(b)\{' \ P \ `\}else\{' \ Q \ `\}',$	$\xrightarrow{\tau^*}$	Q, σ
$\{a \to M\} \cup \{b \to N\} \in \sigma$		

Table 4.3: Some concrete Java evolution rules

There also is another difference between JavaSPI and concrete Java code, regarding data types: the concrete Java code, in fact, does not directly use the cryptographic library **spiWrapper** provided by the framework, but it relies on a custom library that wraps all the types of **spiWrapper** library by additionally defining the serialization strategies that will be used when transferring data through the communication channels. These libraries are formerly called "Marshaling layers" and they do not affect the execution flow of the code. The only case in which a Marshalling layer may affect the execution flow of the code regards the case in which different libraries are used to send and to receive data. However, from the practical point of view, this is a type safety problem and thus it does not need to be handled, as this requirement falls in the "well-formedness" assumption that has already been described.

Finally, a last difference between JavaSPI and concrete Java regards the fact that, in a real Java application, the possible presence of an attacker must be taken in consideration. By relying on the Dolev-Yao attacker model[DY83a] an attacker is a process, executed in parallel to all the other protocol actors, whose knowledge base contains all the public variables and all the data transmitted through public channels. The attacker is able to combine and elaborate information at its disposal by using the same cryptographic tools used by the protocol actors and it can alter the environment by hiding, altering or forging messages. For this reason Java evolution rules cannot guarantee anymore that, anytime a protocol actor sends some data through a public channel, another actor of the protocol is able to receive the same data.

A subset of all the evolution rules is presented in Table 4.3. As usual, for the complete evolution table, please refer to Appendix A.2.

4.1.3 The ProVerif Code

The language used to build models compatible with the ProVerif model checker is a variation of π-calculus. The formal syntax and semantics of this language has already been formally defined in:[Bla09b] the presented operational semantics is very similar to the one described in the previous sections, as it defines groups of statement sets and evolution rules used to process one of the statements of one of the sets. In some particular cases, a single evolution rule is also able to process statements

$`let\ v = SymDecrypt(a,k)\ in\ (' P ')else(' Q ')' \quad \{a \to \{M\}K\} \cup \{k \to K\} \in \sigma$	$\xrightarrow{\tau}$	$P\{v \to M\} \cup \sigma$
$`let\ v = SymDecrypt(a,k)\ in\ (' P ')else(' Q ')' \quad \{a \to \{M\}K\} \cup \{k \to K'\} \in \sigma$	$\xrightarrow{\tau}$	$Q\sigma$
$`if\ a = b\ then\ (' P ')else(' Q ')' \quad \{a \to M\} \cup \{b \to M\} \in \sigma$	$\xrightarrow{\tau}$	$P\sigma$
$`if\ a = b\ then\ (' P ')else(' Q ')' \quad \{a \to M\} \cup \{b \to N\} \in \sigma$	$\xrightarrow{\tau}$	$Q\sigma$

Table 4.4: Some π-calculus evolution rules

from multiple sets (for instance, in the case of a data transfer). However, for what concerns memory handling, in π-calculus, there is no concept of "memory", but the paper presents a renaming function that, anytime a variable is defined, it replaces that variable name with its corresponding value in the rest of the code.

Semantically it is trivial to prove that a series of renaming rules are functionally equivalent to the partial function σ, assigning a variable name to its value: for this reason in Table 4.4 evolution rules of π-calculus are presented by using the same σ function to represent the memory. All the evolution rules presented in this paper use the same formalism, and this greatly simplifies the definition of translation functions in the next sections without representing an important change of the model definition.

The concept of environment and τ/λ evolution rules is preserved in this model, as well. Using a LTS system to define evolution rules is a different technique with respect to the π-calculus formalized in,[Bla09b] where data transfers are formalized as "compound" evolution rules in which two processes evolve at the same time. However, semantical equivalence between the two representations has already been proven in[PS10, PS12] for a slightly modified version of *pi*-calculus called Spi calculus and, moreover, even from an informal point of view is easy to see that the only difference between the two categories of evolution rules just regard how to model the attacker: by using an LTS the attacker can be informally defined as an entity able to manipulate the environment, while by using compound evolution rules the attacker becomes a process P, able to perform any possible data manipulation.

4.2 Translation rules

The JavaSPI framework is able to take a JavaSPI model as input to generate ProVerif models and concrete Java implementations of that model. Formally, these two transformations can be modeled by two translation functions, $J()$ and $PV()$, able to take a set of JavaSPI statements and to generate a corresponding set of statements in concrete Java or π-calculus syntax.

Any well-formed JavaSPI model can be translated by using $J()$ and $PV()$ functions. However, some translation rules need a precise order of JavaSPI statements to be applied, and this order is not forced by any JavaSPI syntax limit: instead, a theorem is presented to prove that sometimes it is possible to change the order of some JavaSPI statements to comply to the $J()$ and $PV()$ translation functions.

Theorem: given a well-formed JavaSPI protocol, it is possible to generate a functionally equivalent version of the same protocol by changing the order of some statements as long as these conditions hold:

(i) The new version of the protocol is still well-formed.

(ii) The new version of the protocol still respects JavaSPI syntax limits.

(iii) The order of statements involving data transmission through a channel and conditional statements is preserved.

Proof: a statement "reordering" operation can be formally defined as an unbounded sequence of swaps of pairs of consecutive statements. In the following proof the swap of statements a and b through P is referred as ($a\ b\ \in P \rightarrow b\ a\ \in P$). As long as the theorem holds for these local swaps it is possible to extend it to any other more complex reordering, as any complex reordering can be represented as a sequence of local swaps.

In JavaSPI it is possible to find 5 categories of statements: variable declarations, variable initializations, data transmissions, conditional statements and custom method calls. Here follows the proof that the theorem holds for all the 5 categories. Key aspect of this proof is the JavaSPI syntax limit that variables must always declared as **final**: for this reason, after its initialization, a variable cannot change its value until it's destroyed.

- a is a method call; this case is impossible, as a can never be a custom method call. By definition, in fact, a custom method call must be the last statement in a process, and implying that a statement b exists after a method call violates well-formedness of the model. Custom method calls are the only statements that can never be moved from its initial position at the end of the code and for this reason will not be considered in the next cases, as any swap involving a custom method call is forbidden by Precondition i: the code must be well-formed before and after the swap.
- a is a variable declaration; here the code remains functionally equivalent to the previous one as long as b is not the instruction initializing or using the variable declared in a. However, this case is forbidden as initializing (or using) a variable before its declaration violates well-formedness of the code. In any other case nothing changes from the functional point of view, as JavaSPI syntax limits impose that the scope of the variable will always range from the position of a to the end of the code. In practice, if a is a variable declaration it can be moved across the code as long as it remains before its initialization.
- a is a variable initialization; this instruction can be moved as long as b is not a statement that reads the value of a: in that case the well-formedness of the code will be violated again, as a final variable cannot be read until it is initialized. In any other case functional equivalence is trivial to prove as, in practice, we move a variable initialization across statements that does not need to read its value. Performance changes may be noted if b is a conditional statement (please note that, if b has both the **if...else** branches, a will have to be placed at the beginning of both branches), but apart from the performance change there are still no functional differences.
- a is a data transmission statement; here acts Precondition iii: b cannot be a conditional statement or another data transmission. This implies that b can only be a variable declaration or initialization, and as variables are read-only it is impossible for b to change the behavior of the data transmission, thus also in this case the movement does not cause any functional difference (provided that the movement respects Preconditions and JavaSPI syntax limits.
- a is a conditional statement; the same reasoning performed in the previous case applies here. Precondition iii limits the nature of b: it can only be an initialization or a declaration of a variable that a is not using in its condition, as initially the code is well-formed and a was before b. For this reason bringing b outside the conditional statement may only alter performance, as before this swap b could have not been processed as often as after the swap, but the functional behavior of the model is unaltered, as b may only add a variable to σ but it will never be able to change the sequence of data transmissions the model will have during evolution.

In conclusion, theorem preconditions are very tight and already limit possibilities of changing statement order. However, as long as a movement is allowed by theorem Preconditions, the generated code has to be functionally equivalent to its initial version, as it does not change the behavior of the system in terms of execution traces. □

$J(`final\ Type\ t = new\ Type(data);'\ P)$	$\rightarrow$	$`final\ TypeCC\ t = new\ TypeCC(data, params);'\ J(P)$
$J(`final\ Pair < A, B > p = new\ Pair(a, b);'\ P)$	$\rightarrow$	$`final\ PairCC\ p = new\ PairCC(a, b)'\ J(P)$
$J(`final\ Hashing\ h = new\ Hashing(a);'\ P)$	$\rightarrow$	$`final\ HashingCC\ h = new\ SubtypeHashingCC\ (a, params);'\ J(P)$
$J(`final\ SharedKey\ sk = new\ SharedKey(a);'\ P)$	$\rightarrow$	$`final\ SharedKeyCC\ sk = new\ SharedKeyCC(a, params);'\ J(P)$

Table 4.5: A portion of the formal definition of the $J()$ translation function

4.2.1 The $J()$ function: from JavaSPI to concrete Java

In order to translate a JavaSPI model to a concrete Java implementation the $J()$ function can be used. Formally, this function takes a JavaSPI execution state as input and produces an equivalent execution state in the concrete Java syntax. This function operates both on the statement syntax and to the memory representation: for what concerns the syntax, every call to the `spiWrapperSim` library is replaced by a call to the `spiWrapper` library, by adding to the call information about the marshaling layer and other implementation parameters. A similar transformation is performed to the memory representation: each cryptographic operation is enriched with the implementation parameters used to perform it.

Depending on the information added to the execution state, it is possible to define infinite $J_{cc,param}()$ functions: CC represents the token used to define the marshaling layer, while $param$ represents the sets of parameters added to every cryptographic operation. In theory, different parameters can be used to translate each statement, even if this could practically make the translated code completely useless (some data encrypted by using certain parameters cannot be decrypted unless using exactly the same parameters). In order to take in consideration all the possible cases, however, the generic $J()$ function does not precisely define CC and $param$: supposing that two execution states P and P' are translated by $J()$, $J(P) = P^J_{cc,param}$ and $J(P') = P'^J_{cc',param'}$. During execution the algorithm cannot assume that $cc = cc'$ and $param = param'$, but it must consider all the possible cases.

4.2.2 The $PV()$ function: from JavaSPI to ProVerif

In this section the PV transition function is defined. This function is able to transform a statement in JavaSPI syntax to an equivalent statement in ProVerif syntax, by also preserving its logical meaning.

As both JavaSPI and ProVerif share the same level of abstraction, this translation is pretty straightforward: it basically just consists in a syntax change for what concerns statements, while memory is unchanged.

$PV(`final\ Type\ t = new\ Type(data);'\ P)$	$\rightarrow$	$`new\ t;'\ PV(P)$
$PV(`final\ Pair < A, B > p = new\ Pair(a, b);'\ P)$	$\rightarrow$	$`let\ p = (a, b)\ in'\ PV(P)$
$PV(`final\ Type\ t = x.decrypt(k);'\ P), \{x \rightarrow \{m\}'_k\} \in \delta$	$\rightarrow$	$`let\ t = SymDecrypt(k, x) in'\ PV(P)$

Table 4.6: A portion of the formal definition of the $PV()$ translation function

4.3 Soundness theorem

Main goal of this section is to prove that any security property proven in ProVerif also holds for the concrete Java code. As ProVerif model checker uses execution traces to prove that a security property is violated, this implies that when a security property is proven in ProVerif it means that, regardless from the choices performed during execution, it is not possible to make the ProVerif model to evolve in a configuration that violates that particular security property. As the evolution rules are deterministic, only one factor can influence how the ProVerif model can evolve: the data exchanged through the channels. For this reason, given one of the actors of the protocol, its "execution trace" represents the pattern of data that it sends and receives from all its communication channels. More formally, an execution trace is a sequence of λ evolution rules that can be obtained by making an actor of the protocol to run against an ideal attacker model able to send data by using all the cryptographic functions at its disposal over its knowledge base, composed on the initialization data and all the data that is sent through "public" and "unsafe" communication channels.

As the same concept of execution trace can be applied to a concrete Java implementation, if all the possible execution traces of a Java application just represents a subset of all the possible execution traces that can be obtained by making its corresponding ProVerif model evolve it is possible to state that, if an execution trace able to violate a certain security property exists in the Java implementation, it also have to exist in the ProVerif model, and the model checker should be able to find it. Otherwise, if the model checker does not find any execution trace able to violate a certain security property, since ProVerif execution traces are a superset of the concrete Java execution traces, it means that it also the concrete Java implementation is immune to that particular type of attack.

Claiming that ProVerif execution traces are a superset of concrete Java execution traces is called defining a "simulation" relation between the two, and it is possible to express it mathematically as $S(P_{PV}\sigma_{PV}, P_J\sigma_J)$, where $P_{PV}\sigma_{PV}$ is the ProVerif model while $P_J\sigma_J$ is the concrete Java implementation.

Since $P_{PV}\sigma_{PV}$ and $P_J\sigma_J$ are not independent each other but they are both generated by applying $PV()$ and $J()$ translation functions to an initial JavaSPI model $P\sigma$, the concept of "Simulation-Generator" $SG(P\sigma)$ is defined: $SG(P\sigma) \implies S(PV(P\sigma), J(P\sigma))$. This is just a compact notation form able to relate the simulation relation to the initial JavaSPI model and to the $J()$ and $PV()$ translation functions, and it enables to define the simulation theorem as follows.

Theorem: for every well-formed JavaSPI model $P\sigma$ that respects all the syntax limits of the language it holds the "Simulation-Generator" property $SG(P\sigma)$, where the $SG(P\sigma)$ property is formally defined with the following formula.

$$SG(P\sigma) = J(P\sigma) \overset{\tau^*}{\rightarrow}\overset{\lambda}{\rightarrow}\overset{\tau^*}{\rightarrow} P'_J\sigma'_J \implies PV(P\sigma) \overset{\tau^*}{\rightarrow}\overset{\lambda}{\rightarrow}\overset{\tau^*}{\rightarrow} P'_{PV}\lambda'_{PV} \wedge$$

$$\exists P'\sigma' | J(P'\sigma') = P'_J\sigma'_J \wedge PV(P'\sigma') = P'_{PV}\sigma'_{PV} \wedge SG(P'\sigma')$$

Informally, the formula states that for each possible execution trace of $J(P\sigma)$ in the concrete Java domain, the same trace also exists in the ProVerif domain. Thus, if a security property is proved in the ProVerif domain, this means that all its traces are safe and, as a consequence, also the subset of traces generated by the Java implementation are safe as well. For this reason, proving that every JavaSPI model is a Simulation-Generator implies proving there is a simulation relation between concrete Java and ProVerif model, thus proving as well the soundness between the ProVerif model and the Java implementation.

Proof: let initially assume that the attacker does not modify execution traces and that the concrete Java implementation is built in a totally automatic way, by adidng the same CC and *param* data to all the statemens. Under these assumptions proving that every concrete Java execution trace is also present in the ProVerif model is trivial, as by definition the evolution tables provided for the three languages are shaped so that if a state $P\sigma$ can evolve to a state $P'\sigma'$ through a series of rules involving at least one λ transition, also $J(P\sigma)$ and $PV(P\sigma)$ have to evolve through a series of rules involving a λ transition labeled by the same symbol. Evolution rules and translation tables are specifically defined to guarantee that the following property automatically holds, in the absence of an attacker:

$$P\sigma \xrightarrow{\tau*}\xrightarrow{\lambda}\xrightarrow{\tau*} P'\sigma' \implies$$
$$(J(P\sigma) \xrightarrow{\tau*}\xrightarrow{\lambda'}\xrightarrow{\tau*} P'_J\sigma'_J \wedge PV(P\sigma) \xrightarrow{\tau*}\xrightarrow{\lambda''}\xrightarrow{\tau*} P'_{PV}\sigma'_{PV} \wedge \lambda = \lambda' = \lambda'')$$

This assertion is possible as $J()$ and $PV()$ functions are modeled to assure functional equivalence between the models. In this case the relation between the states $P\sigma$ and $P'\sigma'$ in the theorem is simply $P\sigma \xrightarrow{\tau*}\xrightarrow{\lambda}\xrightarrow{\tau*} P'\sigma'$. In this trivial case, the theorem is automatically proven by design. □

Let us now also consider additional problematics, such as the presence of an attacker: it may generate several new possible execution traces, as the attacker cam alter data during transmission, thus altering the behavior of the model. However, both ProVerif and concrete Java execution models already consider this eventuality and evolution rules state that both models wukk evolve in the same way provided that the attackes acts in the same way to the two models. Only the JavaSPI model does not consider the attacker presence: for this reason, there can be a divergence between JavaSPI execution traces and the other models, when an attacker acts.

However, as the attacker can only transform data by using the same functions the other actors have at their disposal, it means that it still exist a single JavaSPI state $P^A\sigma^A$ able to generate ProVerif and Concrete java states obtained after a λ evolution rule where data has been forged by a third party. Formally, this can be expressed in the form of the ω transition:

$$PV(P\sigma) \xrightarrow{\lambda A} P^A_{PV}\sigma^A_{PV} \wedge J(P\sigma) \xrightarrow{\lambda A} P^A_J\sigma^A_J \implies$$
$$P\sigma \xrightarrow{\omega} P'\sigma' \wedge PV(P'\sigma') = P^A_{PV}\sigma^A_{PV} \wedge J(P'\sigma') = P^A_J\sigma^A_J$$

The ω evolution rule is a totally artifical evolution function that can be used to relate $P\sigma$ to $P'\sigma'$, thus representing the action of an attacker in the JavaSPI model. This rule is defined as follows:

$$`Type\ v = c.receive(Type.class)'P, \sigma \xrightarrow{\omega} P, \{v \to X\} \in \sigma$$

It basically states that data received from a channel may be arbitrary chosen disregarding from the sender process, which may or may not even exist. In the case of an attack, then, it is still possible to prove the soundness theorem by stating that the function relating $P\sigma$ and $P'\sigma'$ is the following:

$$P\sigma \xrightarrow{\tau*} \xrightarrow{\omega} \xrightarrow{\tau*} P'\sigma'$$

The ω function does not need to exist in reality, and it does not model any real behavior of a JavaSPI application: soundness relation between ProVerif and the JavaSPI model is not needed, in fact, and thus it does not matter if JavaSPI does not have the same execution traces of Java and ProVerif. However, ω evolution rules prove the existence of the hypotetical JavaSPI state $P'\sigma'$ able to make the soundness theorem hold when an attacker alters communication. □

Finally, the last assumption performed at the beginning of the proof must be lifted: it is needed to consider the case in which some implementation details differ between two concrete Java statements: for instance, actor A may calculate hash codes with the MD5 algorithm, while actor B calculates hash codes with the SHA-1 algorithm. This difference may cause the failure of some statements in the concrete Java code, and for what has been told up now the JavaSPI and ProVerif models cannot map the same behavior, as they have no notion about hashing algorithms. This problem must be handled, otherwise it may break the entire simulation relation, as it makes impossible to state that concrete Java execution traces are a subset of the ProVerif model execution traces.

This problem has been solved by shaping the $PV()$ translation function so that, every time a conditional statement must be translated in ProVerif, an additional conditional statement is generated: this additional condition is under the control of the attacker that can decide how that condition should evolve. Informally this means that, in ProVerif, development errors are mapped to malicious behaviors performed by the attacker model: in this way all the possible Java execution traces have an equivalent representation in the ProVerif model, even by considering development issues.

In a similar way to the previous case, once it can be proven that concrete Java execution traces are a subset of ProVerif execution traces, eventual divergences between JavaSPI and the other models are mapped by defining artificial JavaSPI evolution rules that does not map any real JavaSPI behavior but that are able to provide a new execution state that is aligned to ProVerif and concrete Java execution states:

$$`if(condition)\{'P`\}', \sigma \xrightarrow{\omega} 0, \sigma$$
$$`if(condition)\{'P`\} else \{'Q`\}'\sigma \xrightarrow{\omega} Q, \sigma$$

Two new artificial ω evolution rules are forged to prove the existence of the $P'\sigma'$ state that allows to make the soundness theorem hold also in this case.

$$P\sigma \xrightarrow{\tau*} \xrightarrow{\omega} \xrightarrow{\tau*} P'\sigma'$$

□

As all the assertions performed in this section can be proven for each possible piece of JavaSPI code by just combining together evolution and translation rules, this implies that Soundness-Generator property holds for any JavaSPI piece of code, thus it is possible to infer that each security property proven by ProVerif can be extended to the corresponding concrete Java code, as long as ProVerif and Java are both generated by applying the $J()$ and $PV()$ translation functions to the same well-formed JavaSPI code ($P\sigma$).

4.4 Syntactical extensions

The JavaSPI syntax described in the previous sections is heavily limited: variables can only be declared as "final" and it is not possible to write code after an "if...else" statement. Moreover, being limited to the SpiWrapperSim data type library makes every complex data type to be extremely verbose to be defined, as the only structure able to compose different pieces of data is the "Pair" class.

These limits have been posed to bring the JavaSPI syntax as close as possible to the Applied π-Calculus syntax used in ProVerif models: this greatly simplified proving the soundness theorem, but it heavily impacts to the expressiveness freedom of developers that are using JavaSPI to develop models.

Several of these limits can be lifted, at least in theory, by preserving the soundness, but proving it would make the proof overcomplicated. For this reason, rather than modifying the model, it has been decided to develop a code pre-processor that is able to transform a piece of JavaSPI code not respecting all the limits described in the previous sections in a functionally equivalent piece of code compliant to all the previously defined specifications. This pre-processor does not alter the JavaSPI model, in this way, thus providing syntax extensions without breaking the soundness relation previously proved.

Here follows an informal description of the pre-processing translation rules:

- *Non-final variables*: the pre-processor allows to transform non-final variables to final ones. Every time a variable is re-assigned, the statement that re-assigns the variable is replaced with a statement creating a new variable with a different name. Then, all the next references to that variable are replaced with the new variable name.
- *Code after a conditional statement*: during pre-processing, all the code written after a conditional statement is moved inside the conditional statement itself and, eventually, it is dupicated among the "if" and the "else" branches. More formally, $`if(cond)\{'P`\}else\{'Q`\}R' \rightarrow `if(cond)\{'PR`\}else\{'QR`\}'$. Even if this operation trivially makes a piece of code compliant to JavaSPI specifications, it may enormously increase the size of the generated Application/Model, thus it should be used with caution.
- *Code after a custom method call*: the translation is very similar to the previous case: the block of code written after the custom method call is moved inside the method itself. However, in this particular case, method parameters are also adjusted so that all the variables used in the moved piece of code are still accessible.
- *Custom data types*: some new classes have been added to the SpiWrapperSim library, 'List' and 'Packet'. The first one can be used to create bounded lists of objects (all of the same data type), while the second one is an abstract class that can be used to create custom objects containing a collection of heterogeneous data types. Both the objects are translated into a composition of nested pairs during pre-processing. However, differently from other syntactic sugar transformations, in this case the Packet semantic is preserved in Java concrete code, through a post-processing translation function that re-transforms the nested pairs into concrete, automatically-generated, custom data types. From the functional point of view there is no practical difference, as the generated custom data types are shaped to guarantee its functional equivalence with the nested pairs. However, from the practical point of view, using these custom data types have several advantages: for instance, it greatly simplifies developing the serialization layers when data types are particularly complex. Moreover, avoiding nested pairs makes the concrete protocol code much more compact and understandable.

5

Case studies

Using JavaSPI to model a particular configuration of the SSL v3.0 handshake protocol and using it to build an implementation able to interoperate with commercially-available implementations of the same protocol, like the openssl tool.

Contents

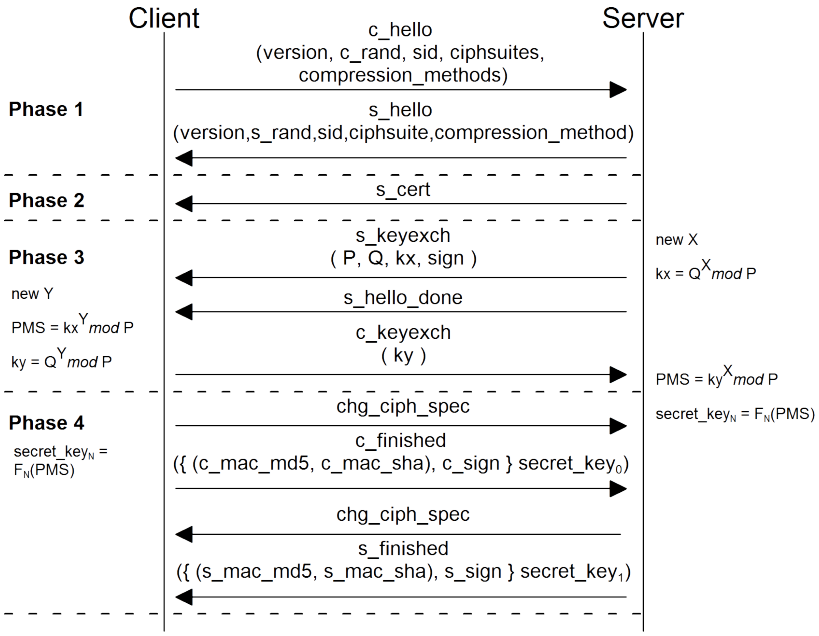

Figure 5.1: SSL message exchange in the selected scenario.

5.1 SSL 3.0

In order to provide a validation example of the proposed JavaSPI approach, a simplified but interoperable implementation of both the client and server sides of the SSL handshake protocol has been developed.

The considered scenario, depicted in Figure 5.1, can be logically divided into four different phases:

1. Client and server exchange two "hello" messages which are used to negotiate protocol version and ciphersuites.
2. The server authenticates itself to the client by sending its certificate `s_cert`.
3. Diffie-Hellman (DH) key exchange is performed; note that the server DH parameters are signed by the server.
4. Finally, the session is completed by the exchange of encrypted "Finished" messages.

For simplicity, in the considered scenario both the developed client and server only support version 3.0 of the protocol with DSA server certificate. Other ciphersuites or other protocol features such as session resumption or client authentication are not considered. Indeed, the goal is to validate the methodology with a minimal, yet significant example, rather than provide a full reference implementation of the SSL protocol.

The SpiWrapperSim library has been used to develop the abstract model of the SSL protocol. This includes eight new Packet classes representing the structures of the different types of exchanged messages and a client and a server SpiProcess classes. In addition, an "instancer" process called *Master* that just runs an instance of client and server in parallel has been added in order to simulate protocol execution. Figure 5.2 provides a code excerpt of the Java SSL model.

After defining the model the following properties have been expressed and successfully verified:

- Secrecy of the client and server DH secret values.

Server.java

```
class Server extends SpiProcess { ...
 @Override void doRun(final Channel c,
   final Identifier SSL_VERSION_3_0,...)
 { ...
   final Pair<Identifier, DHHashing>
      c_key_exch = c.receive(Pair.class);
   final DHHashing c_DHy = c_key_exch.getRight();
   final Triplet PMSp =
      new Triplet(c_DHy, DH_x, DH_P);
   final DHHashing common_key =
      new DHHashing(PMSp);
 }
}
```

Master.java

```
class Master extends SpiProcess {
 @Override void doRun()
 { ...
   final Client c = new Client(...);
   final Server s = new Server(...);
   start(c,s);
 }
}
```

Figure 5.2: An excerpt of the SSL protocol abstract model.

```
@SharedKeyA(Algo="3DES", Strength="168")
@SharedKeyCipheredA(Algo="3DES", Mode="CBC")
public class Server extends spiProcess {
...
final Hashing c_write_iv = new Hashing(PA3);
...
@Iv(type=Types.varName, value="c_write_iv")
final SharedKeyCiphered
  <Pair<Pair<Hashing, Hashing>, Hashing>>
  c_encrypted_Finish =
     c.receive(SharedKeyCiphered.class);
```

Figure 5.3: An excerpt of the Java model with annotations on it.

- Server authentication, expressed as an injective correspondence between the correct termination of the two processes: each time a client correctly terminates a session, agreeing on all relevant session data and the server identity, a server must have started a session, agreeing on the same session data and the server identity.

Finally, in order to grant interoperability, a custom marshaling library compliant with the SSL standard has been developed.

Besides setting the marshaling layer, it was also necessary to specify by annotations the needed cryptographic details, such as algorithms and related parameters. In the sample SSL protocol both compile time and run time resolution features of JavaSPI have been exploited. Even if this protocol implementation uses many "hardcoded" parameters, like the ciphersuites and the key strengths, other information is only known at run time: for example, the initialization vectors used for shared key encryption are calculated from the shared secret, thus they change at each run.

As shown by the code excerpt in Figure 5.3, any static detail can be specified once, on the head of the class, while the dynamic details and the special cases are specified in front of each variable that needs them. In the sample code, the initialization vector is computed by applying a hash function and is stored in variable `c_write_iv`. Then, an annotation specifies that the initialization vector for the ciphered message received in variable `c_encrypted_Finish` is the value in variable `c_write_iv`.

The amount of required annotations does not burden the code: the SSL example required about 60 annotations in total (client + server), that amounts to about 10% of the whole model size. To make this measure significant, few default values have been used; in other words, default values where not crafted to suit the SSL example.

The generated client and server implementation have been successfully tested for interoperability against OpenSSL 0.9.8o.

5.1.1 Performance considerations

One claimed disadvantage of code generation techniques is that as the code is automatically generated it will never be as optimized as it is possible to do by manually writing the code. Nonetheless, with cryptographic protocols it is often the case that the main computing effort lies in the computation of cryptography: for this reason the possible overhead due to potential code inefficiency is often negligible with respect to the overall computing time.

In order to experimentally confirm this claim, we compared the performance of the SSL client implementation generated with JavaSPI to the performance of a corresponding code into the JSSE library, which is the Oracle's Java official implementation of SSL. The two codes have been executed against the same SSL server, based on the OpenSSL application.To ensure that the two clients are effectively performing the same operations, a custom Certificate validator has been written for the JSSE implementation in order to treat the certificates in the same way they are treated by the JavaSPI SSL implementation. As a further check, some network packet sniffing has been preliminarily performed in order to ensure that the same ciphersuites were used, and the same messages were exchanged. Finally, in order to run the two applications in the same environment and limit random components in the measurements, the tests were run keeping every communication local, thus eliminating random network latencies. Moreover, the two implementationswere run in the same Java virtual machine a thousand times and the average execution time and its standard deviation were computed. Since in the first run a Java program is affected by the Java class loader latency, the time of the first run has been excluded, while all other measurements have been used to compute average and standard deviation values.

The resulting graph, shown in Figure 5.4, reports the obtained results: as it is possible to observe, the processing times of the two implementations are nearly the same; there is just a performance difference of less than 5% between the two implementations. As stated before, the explanation is that both the pieces of code are using exactly the same cryptographic library (JCA) and the DSA signature check and DH modular exponentiation performed in the SSL protocol take the main part of the total protocol execution time. It is likely that the JSSE implementation is much more optimized than the JavaSPI auto-generated code, but this performance boost just affects a very small portion of the total execution time. In conclusion, the performance results show us a very small difference between an optimized version of the code, written by hand, and an automatically generated implementation. This inefficiency might be considered non negligible in some particular cases, but in any other situation having an implementation with an high level of trustworthiness and correctness can greatly balance this small performance penalty.

5.1.2 Results

The JavaSPI framework enables model-driven development of security protocols based on formal methods without the need to know specialized formal languages. Knowledge of a formal language is

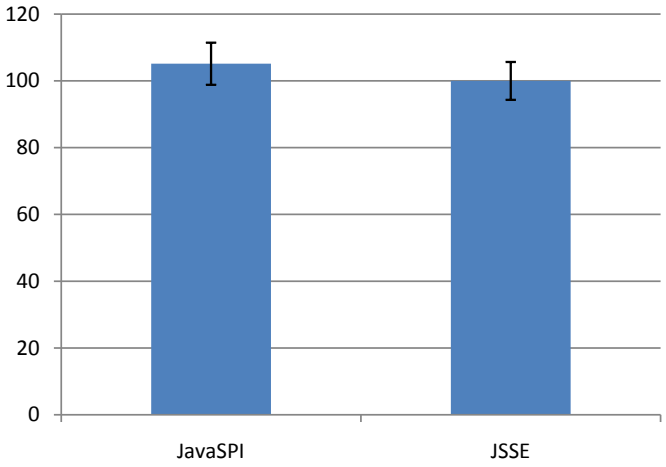

Figure 5.4: Timing comparison between the handshake performed by JavaSPI SSL client implementation and JSSE's one

replaced by knowledge of a Java library and of a set of language restrictions, easier to learn by Java experienced programmers. Moreover, standard IDEs can be used to develop the Java model, with the benefit of having access to all the development features offered by such IDEs, like debugging and code-completion.

The proposed approach, along with the provided toolchain and libraries, enables:

(i) interactive simulation and debugging of the Java model, via standard Java debuggers available in all common IDEs;

(ii) automatic verification of the protocol security properties, via the de-facto standard ProVerif tool;

(iii) automatic generation of interoperable implementation code, via a custom tool, driven by Java annotations embedded into the model files.

Compared to similar frameworks, like Spi2Java, JavaSPI is easier to use, while retaining the nice feature of enabling fast development of protocol implementations with high integrity assurance given by the linkage between Java code and verified formal models. Future work includes focusing on the formalization of the relationship between Java and spi calculus semantics, in order to get a soundness proof for the Java code, once the ProVerif model is verified. From an engineering point of view, porting the ProVerif verification results directly to the Java model could further improve usability and accessibility of the proposed framework. Moreover, further tests could be performed in order to demonstrate that quite every Java developer is able to design and validate a communication protocol by just reading the framework documentation.

Part II

Enforcing security through traffic monitoring

6

Background

Background information regarding Finite State Automata, FSA optimization techniques and typical traffic analysis deployment strategies

Contents

As anticipated in the introductory sections, this part of the thesis proposes new algorithms and techniques able to make traffic analysis more "affordable" to the average user, either by speeding up traffic analysis through general purpose devices, in order to make their adoption feasible in place of using expensive special-purpose hardware, and by proposing novel ways to intelligently take advantage of existing unused resources in order to further reduce the workload posed on a single machine.

However, in order to be able to understand the next chapters, it is necessary to provide some background information about the mathematical tools that are being used both to perform traffic processing and to accelerate it.

For this reason, this chapter is structured as follows: at first the mathematical definition of Finite State Automata (in particular in its Nondeterministic form, called NFA) is presented in Section 6.1, then Section 6.2 shows how the iNFAnt tool is able to use these NFAs to process data and spot the presence of data patterns matching a certain regular expression. Then, Section 6.3 describes one of the most popular techniques used to accelerate Finite State Automata processing by building equivalent but more complex representations of them. Finally, Section 6.4 describes how a typical traffic analysis algorithm is deployed in a network, and why it is usually shaped as a series of stream processing blocks.

Table 6.1 lists the main symbols used in this chapter along with a short description of their meaning.

Table 6.1: Common symbols and notation used in this chapter

Q	Set of states of an NFA
δ	Set of transitions of an NFA
Σ	Set of symbols of an NFA (alphabet)
A	Set of accepting states of an NFA
$\|X\|$	Number of elements of set X (cardinality)
$\overline{N}_{fo}$	Average number of states of an NFA that can be reached starting from a single state, by following all its outgoing transitions (average fan-out)
$\overline{L}_s$	Average number of transitions of an NFA connecting the same pair of states (average label size)
$\overline{R}$	Average number of transition ranges of an NFA connecting the same pair of states (similar to $\overline{L}_s$ but it counts *ranges* instead of each single transition)

6.1 String matching through FASs

One of the main advantages of using regular expressions to define complex patterns of text is that is always possible to transform them into an equivalent Finite State Automata (FSA), which provides mathematical foundation and enables the use of simple algorithms to handle regular expressions.

For instance, simple algorithms exist to determine if a particular input matches a regular expression represented by a given FSA and for the composition of multiple FSA. The latter operation is typically used when complex rule sets, such as the made of thousands of regular expressions used by NIDS, have to be checked within a single scan of the input.

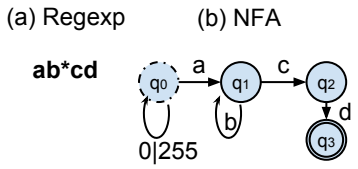

Figure 6.1: A simple regular expression in its (a) "textual" representation and (b) NFA form

FSAs come in two different forms with equivalent expressiveness, namely Deterministic FSA (DFA) and Non-deterministic FSA (NFA). A DFA guarantees a bounded execution time for the matching operation on any execution architecture, because for each input string a single path in the automaton has to be followed, with a single state traversal and a single memory access for each symbol of the input string. However, the bounded execution time is achieved at the expense of a potentially huge memory consumption, particularly when complex regular expressions (e.g., with frequent use of repetition wildcards ".*") are used, or when many expressions are combined together in a single DFA. While this problem can be alleviated by using variations of the DFA, important limitations still exist on the number and the complexity of the rule sets that can be handled using these representations. This may force application developers to use approximate (and simpler) regular expressions, which, however, may either impair the capability to filter out exactly the traffic one is interested in, or potentially generate many false positives.

The NFA form solves the previous problem because the number of states in the automaton is directly proportional to the length of the regular expressions used to create the FSA. However, this advantage is paid with a significant increase in the execution time of the matching operation, which is no longer predictable and may grow dramatically with the complexity of the automaton. In fact, with a NFA, the matching of a given input may require to follow more than one processing path, whose number is not predictable and depends on the complexity of the automaton and on the particular input data that is being processed. This feature can lead to dramatically high and variable execution times when using strictly sequential algorithms for NFA-based regex matching. For this reason, software-based implementations with strict processing time constraints (such as packet processing applications running on general purpose CPUs) are usually based on the DFA form or on some of its variations.

However, when it is possible to rely on highly parallel hardware architectures, the NFA paradigm becomes suitable, as architectures like a GPU can make the execution time less variable by exploring all the active processing paths in parallel, at no additional expense.

Figure 6.1 presents an example of a simple regular expression with its corresponding NFA and 2-stride NFA. In the figure, the states with the dashed border are the initial states while those with the thick border are the final (i.e. accepting) states. The notation '`0|255`' stands for "any symbol in the range from 0 to 255". This means that the transition with that label is in fact a set of transitions, one for each possible input symbol.

A NFA can be formalized as a 5-tuple $\langle Q, \Sigma, q_0, \delta, A \rangle$ where Q is the set of states, Σ is the set of symbols (the alphabet), $q_0 \in Q$ is the initial state, $\delta \subseteq Q \times \Sigma \times Q$ is the transition function, i.e. a set of transitions, where each transition is a triple (q_1, s, q_2) with initial state $q_1 \in Q$, label $s \in \Sigma$ and final state $q_2 \in Q$, and $A \subseteq Q$ is the set of accepting states.

A NFA takes as input a sequence of symbols $s_1, ..., s_n$, with $s_i \in \Sigma$. A match is found at symbol s_k if there exists a sequence of $k+1$ states $q_0, ..., q_k$ that starts with the initial state and terminates with an acceptance state q_k, and there exists a sequence of transitions labeled $s_1, ..., s_k$ that bind each state of the sequence to the next state, i.e. $\forall i \in [1, k] : (q_{i-1}, s_i, q_i) \in \delta$.

The classical NFA matching sequential algorithm keeps track of all the states that can be reached after each input symbol and reports a match when an accepting state is reached. The algorithm is based on keeping a set of active states that at the beginning includes only the initial state. Input

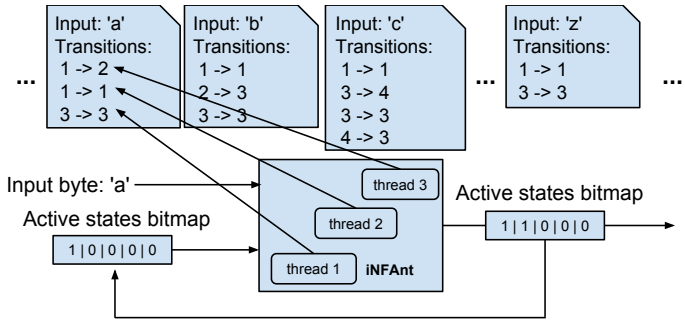

Figure 6.2: Overview of the iNFAnt processing structure

symbols are read sequentially and for each input symbol the next set of active states is computed by following all the enabled transitions, i.e., the ones that fire based on the given input symbol and that start from any of the current active states.

6.2 The iNFAnt string matching processor

iNFAnt[CRRS10] is an efficient NFA-based regex matching processor that runs on GPUs. Its algorithm consists of iteratively reading input symbols and updating the set of active states, represented by a bit vector where each bit corresponds to a state of the NFA. After reading a new input symbol, the algorithm looks for the transitions that can be triggered by that symbol. This operation is made simple by storing transitions already grouped by input symbol. These transitions are then used to update a bit vector that keeps the active states. As shown in Figure 6.2, several transitions are processed in parallel by assigning them to different threads, which contributes to alleviate the dependency of the update time on the number of transitions triggered by the input symbol. This is made possible by exploiting the high number of threads that can run on a GPU and the high memory bandwidth that can be obtained by adopting a clever memory access policy.

In addition to this form of parallelism, iNFAnt can also process several input strings in parallel, by assigning each string to a different group of threads. The scheduler can then exploit the large number of threads to hide memory latency, by scheduling new groups of threads when the current ones are waiting for data from main memory (memory access time is far beyond the processor cycle time), hence keeping the processor busy almost all the time.

Thanks to the two forms of parallelism, iNFAnt has a reasonably stable throughput, which depends on the average number of transitions per symbol. When the number of transitions triggered by a symbol exceeds the maximum number of threads supported by the GPU, the algorithm has to iterate transition processing, hence decreasing throughput.

6.3 Multi-Stride and Alphabet Compression

Even by taking advantage of GPUs, the iNFAnt processing throughput is still limited, just like any other software-based NFA or DFA regexp-based string matching processor. For this reason, research focused on finding additional techniques to futher improve these limits. Multi-striding, for instance, is a technique that transforms an FSA so that it can process more than one input symbol at a time, hence decreasing the number of steps needed to complete the matching of a given input data packet.

For instance, if a DFA takes N steps to perform the matching operation, where N is the length of the input data (e.g., the size of a network packet), a 2-stride version of the DFA returns the same result in $N/2$ steps. With NFAs, similar reductions can be achieved.

Although, in theory, multi-striding can be applied by grouping together an arbitrary number of input symbols, in practice the use of too many symbols leads to an automaton that becomes so complex that prevents the building process to complete in reasonable time (and memory). Therefore, multi-striding is usually implemented by iteratively doubling the stride level until further doubling becomes unfeasible, i.e. first the NFA is transformed into a 2-stride NFA, then the latter is transformed into a 4-stride NFA, and so on.

In addition, while the number of states of a multi-strided automaton does not change with respect to the original FSA, the number of transitions may quadratically increase in the worst case. For this reason, multi-striding is usually coupled with *Alphabet Compression*,[BTC06, BC07] a technique that reduces the number of transitions of the multi-stride automaton by performing a compression of its input alphabet.

6.3.1 Multi-Stride algorithm

Algorithm 1 shows a simplified version of the stride doubling algorithm presented in,[BC08] which can be considered the current state of the art.

Algorithm 1 The stride doubling algorithm of.[BC08]

```
1:  δ' = {}
2:  queue = {q0}; processed = {q0}
3:  while !queue.empty() do
4:      q0 = queue.pop()
5:      for all sA ∈ Σ, q1 ∈ Q | (q0, sA, q1) ∈ δ do
6:          for all sB ∈ Σ, q2 ∈ Q | (q1, sB, q2) ∈ δ do
7:              δ' = δ' ∪ {(q0, (sA, sB), q2)}
8:              if q2 ∉ processed then
9:                  queue.push(q2)
10:                 processed = processed ∪ {q2}
```

The algorithm builds the new set of transitions δ' by enumerating, for each reachable state q_0, all the possible combinations of two consecutive transitions (lines 4-6 in the code). For each of those combinations, the algorithm generates a transition in the new automaton that is equivalent to the two original transitions, i.e. a transition that links directly q_0 to q_2 with a label that is the concatenation of the labels associated with the two original transitions (line 7). The algorithm starts by processing the initial state (line 2) and then it iterates through all the states that can be reached by using the generated compound transitions (lines 8-10).

For simplicity, the presented pseudo-code does not handle the particular case where the intermediate state q_1 is an acceptance state while q_2 is not, but basically the complete algorithm simply solves this problem by adding an extra transition leading to a new state corresponding to q_1.

As, usually, the algorithm processes a number of states equal to the states of the original automaton, the asymptotic time complexity of the algorithm can be evaluated as $|Q| \cdot (\overline{N}_{fo} \cdot \overline{L}_s)^2$ where the meaning of $\overline{N}_{fo}$ and $\overline{L}_s$ is explained in Table 6.1. This formula derives from the observation that, starting from each of the $|Q|$ states of the NFA, it is necessary to iterate through all its $\overline{N}_{fo} \cdot \overline{L}_s$ outgoing transitions and, then, for each reached state, $\overline{N}_{fo} \cdot \overline{L}_s$ compound transitions are added.

6.3.2 Alphabet compression

The necessity of alphabet compression comes from the consideration that the alphabet size $|\Sigma|$ grows exponentially with the stride level (it becomes $|\Sigma|^k$ for the k-stride NFA), and is directly proportional to the $\overline{L}_s$ factor of the stride doubling asymptotic complexity formula. Hence, this growth of the alphabet size impacts not only on the complexity of the resulting NFA but also on the possibility for the stride doubling algorithm to terminate in a reasonable time. As a consequence, alphabet compression is executed after each stride doubling step in order to decrease the cardinality of the new alphabet.

The main idea behind any alphabet compression algorithm is that often there are symbols in the alphabet that are equivalent, i.e. they always trigger *exactly* the same set of transitions. This happens because often the patterns used to scan the network traffic are limited to alphanumeric characters (e.g., '`0-9A-Za-z`') and to a few other symbols, while the rest are ignored. If two (or more) input symbols (e.g., '`aa`' and '`bb`') always originate the same transitions, they are replaced with a single one (e.g., 'α') by alphabet compression, thus decreasing the number of input symbols in the alphabet. In essence, this process enables each new symbol of the new alphabet to represent an entire class of symbols of the original alphabet; as a consequence, the translated NFA that uses this dictionary has a smaller number of transitions than the original one although the two NFAs are functionally equivalent.

Obviously, as alphabet compression changes the alphabet of the NFA, each input string (i.e., each packet) has to be translated to the new alphabet by substituting pairs of consecutive symbols with the new symbols assigned to their equivalence classes. However, as the time required to perform this operation is generally lower than the time required for matching, the overhead of this data translation can be considered negligible. Furthermore, on hardware architectures with multiple execution units, this task can be pipelined with the regular expression matching task.

Figure 6.3 shows the equivalence classes built for a simple 2-stride NFA. The map on the right hand side is a graphical representation of the space of symbol pairs to be partitioned into equivalence classes (each cell in the map represents a single symbol pair). Symbol pairs are partitioned into equivalence classes according to the transitions they can trigger. For example, the equivalence class of symbol pairs that can trigger only a transition from q_0 to q_1 (the area labeled with $q_0 \rightarrow q_1$) is made up of the symbol pairs '`a|f,a|f`' with the exclusion of '`e|f,e|f`'. In fact, the pairs '`e|f,e|f`' trigger two transitions, from q_0 to q_1 and from q_1 to q_2, so that they constitute another equivalence class. The total number of classes is 4: the other 2 classes are made up of the symbol pairs that can trigger only a transition from q_1 to q_2, and those that can trigger no transition (the area of the map not covered by rectangles). Consequently, the new alphabet is made up of only 4 symbols, each one assigned to one equivalence class, and the translation dictionary replaces all the symbols of each equivalence class with the new symbol assigned to that class.

Determining the equivalence classes by building the sets of transitions triggered by each symbol pair is unfeasible with large automata, mainly because of the huge amount of memory required: the size would be $O(|\Sigma|^2|Q|^2)$, as for each cell of the map a set of state pairs should be stored.

A better algorithm was initially proposed in by Brodie at al. in,[BTC06] which can perform the same operation using just one integer and one boolean for each symbol pair but its time complexity is $O(|\Sigma|^4|Q|)$. The algorithm that is currently the state of the art was proposed by Becchi and Crowley in,[BC13] which trades a slight increase in memory consumption (it requires 2 integers and 2 booleans per symbol pair) for an noticeable improvement in time efficiency.

With this algorithm (shown in Algorithm 2) the translation dictionary (an array of integers called `map`) is built in an iterative way that the authors call *cluster division*: initially all the elements of the dictionary are filled with the same value (0), meaning that all the possible symbol pairs are translated

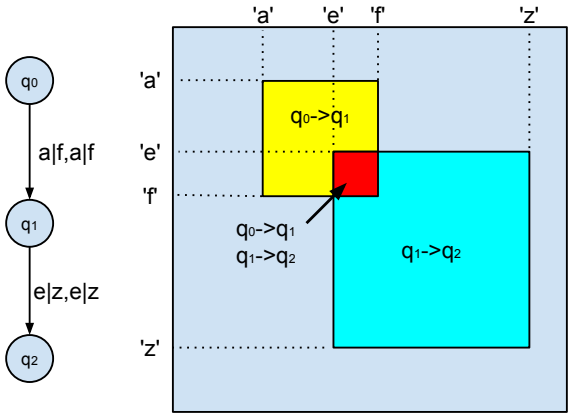

Figure 6.3: An example of alphabet compression

to the same equivalence class, and then it iteratively divides the classes by considering separately each possible combination of states $(q_1, q_2) \in Q \times Q$ (lines 2-3). The main idea of each division step is that a symbol pair '`ab`' has to be remapped to a new class if from q_1 to q_2 there is no transition labeled with it but there are transitions labeled with other symbol pairs that previously belonged to the same class as '`ab`'. The algorithm uses the two arrays of booleans named `char` and `class` to record respectively the set of symbol pairs that label the transitions from q_1 to q_2 and the classes covered by these symbol pairs. A first iteration computes these arrays and a second iteration does the necessary remapping, using a support array of integers named `remap`, which records the already performed remapping operations.

Algorithm 2 The alphabet compression algorithm presented in.[BC13]

```
1:  map[|Σ|][|Σ|] = 0; size = 0
2:  for all q1 ∈ Q do
3:    for all q2 ∈ Q do
4:      char[|Σ|][|Σ|] = false
5:      class[|Σ| × |Σ|] = false
6:      remap[|Σ| × |Σ|] = 0
7:      for all (a, b) ∈ Σ × Σ | (q1, (a, b), q2) ∈ δ do
8:        char[a][b] = true
9:        class[map[a][b]] = true
10:     for all a ∈ Σ do
11:       for all b ∈ Σ do
12:         if !char[a][b] & class[map[a][b]] then
13:           if remap[map[a][b]] = 0 then
14:             remap[map[a][b]] = ++size
15:           map[a][b] = remap[map[a][b]]
```

The asymptotic time complexity of this algorithm can be evaluated as $|Q|^2 \cdot |\Sigma|^2$. The $|\Sigma|^2$ factor is due to the two iterations through the entire array, while the $|Q|^2$ factor comes from the repetition of these iterations for every pair of states. The most critical factor in this formula is $|\Sigma|^2$ because $|\Sigma|$ rapidly increases with the stride level while $|Q|$ is almost constant.

Memory consumption in this algorithm is also a critical issue, as the size of the data structures used by the algorithm grows proportionally to $|\Sigma|^2$.

Due to these limiting factors, reaching high stride levels with big rule sets by using the current state-of-the-art algorithms results unfeasible.

6.4 General structure of Traffic Analysis algorithms

By giving a broader look to the iNFAnt tool it is possible to conceptually split its functioning in two (or more) parts: at first, data has to be translated by using symbols defined during the alphabet compression stage, and this process can be repeated several times depending on how high the Multi-Stride algorithm has been run over the NFA. Then, finally, a set of data streams is grouped in a single packet of data that is sent to the GPU, where the NFA is already loaded in memory and ready to parallely process all the data streams that have been sent. It should be possible to state that iNFAnt is structured as a series of stream processing blocks, each of them using a different execution unit (the CPU to translate data and the GPU to process it) in order to build a processing pipeline.

This structure is not specific to iNFAnt, but it is possible to notice that the main part of traffic analysis algorithms are structured in this way. The main reason for this phenomenon is that the input data arrives as a continuous stream, and often this stream is so huge that it cannot be considered reasonable to temporarily store and process it in a second time. This last constraint makes mandatory to process data in a semi-real-time fashon even in the cases on which immediately obtaining the processing results is not required.

For this reason, a lot of traffic analysis algorithms are implemented as stream processing tools. A common mathematical representation for stream processing algorithms has been reported in[THW02] by means of Directed Acyclic Graphs. However, this representation is slightly limiting to properly represent a wide range of Traffic Analysis Algorithms, since often the various processing blocks communicate each other by means of bi-directional communication channels. For this reason, throughout this thesis we have taken in consideration all the Traffic Analysis algorithms that can be represented by means of “Extended DAGs”, where the extension simply regards these additional non-oriented interconnections used to exchange control information. This concept is important because some of the techniques that will be presented in the next chapters rely on this particular feature of the traffic analysis algortihms.

7

Improving String Matching algorithms

How to improve the NFA optimization techniques presented in the previous chapter?

Contents

Using GPU-accelerated tools like iNFAnt to perform string matching is a very promising technique to replace expensive specialized hardware with general purpose devices and still keep a reasonable processing throughput. However, as explained in the previous chapter, the maximum obtainable throughput is currently limited by the GPU performance, the size of the rule set and the time available to optimize it. This chapter focuses on providing novel approaches able to push forward the last two limits: in particular, Section 7.1 proposes faster algorithms to perform Multi-Stride, while Section 7.2 proposes a novel technique, called Multi-Map Multi-Stride, able to contain the complexity of NFAs after each stride doubling, in order to make them easier to handle and to optimize them again. This simplification comes with the cost of an additional processing overhead, but this overhead can be made negligible by taking advantage of GPUs parallel procesing features. For this reason Section 7.3 describes an optimized version of the iNFAnt architecture making it able to process both Multi-Strided NFAs and Multi-Map Multi-Strided ones in a more efficient way, by minimizing the issues of Multi-Map NFAs to make their adoption viable. The algorithms proposed in this section are an evolution of the basic algorithms presented in.[ARS12, ARSar]

7.1 Accelerating Stride Doubling and Alphabet Compression

This section focuses on providing improved algorithms to perform both stride doubling and alphabet compression operations. From a mathematical point of view, both the algorithms suffer from performance issues when facing big and comlplex NFAs. In particular, it is known that after every stride doubling the alphabet size of the nfa (Σ), increases quadratically ($\Sigma' = \Sigma^2$): the problem is that the best computational complexity formulas of stride doubling and alphabet compression presented in literature up now, like in,[BC08] are both depending on Σ: in the stride doubling formula $|Q| \cdot (\overline{N_{fo}} \cdot \overline{L_s})^2$ the critical component is $\overline{L_s}$, representing the average amount of label symbols connecting each state pairs, that in the worst case can become equal to Σ; for what concerns alphabet compression, instead, the problem is even more serious, as its complexity formula is $|Q|^2 \cdot |\Sigma|^2$. Moreover, especially for what concerns alphabet compression, an additional problem regards the requirements of available memory.

7.1.1 Stride Doubling with Range-Based notation

The asymptotic time complexity of the state of the art stride-doubling algorithm depends mostly on the number of compound transitions to be generated, which tends to increase rapidly at each stride doubling. Hence, the time required for stride doubling becomes unreasonably large even after the first doubling.

In order to reduce the impact of this problem, we change how transitions are represented: transitions are grouped according to their source and destination states and the labels of the transitions of each group are stored by using *ranges* instead of individual symbols. For example, if we consider an NFA with the transitions that link q_0 to q_1 labeled by the symbols '`97, 98, ..., 121, 122`', this set of labels would be specified by storing only the first and the last labels, i.e. '`97|122`'. When the values are not completely contiguous, more ranges are used.

The range notation is exploited in the stride doubling algorithm by treating ranges as atomic entities during the creation of compound transitions: rather than iterating on every possible symbol of each label set, the algorithm just iterates on ranges and it creates compound transitions just by combining ranges together. For example, a transition set made up of the ranges '`97|122`' and '`200|200`', when combined with another set containing only the range '`50|100`, will generate a combined set of

transitions labeled with two pairs of ranges: '`97|122,50|100`' and '`200|200,50|100`'. Building the compound transitions by working with ranges produces only two transition sets and requires only two iterations of the algorithm, compared to the huge number of iterations (equal to the number of symbol pairs) with the traditional algorithm[i].

By introducing the range notation, the asymptotic time complexity of stride doubling changes from $|Q|(\overline{N}_{fo} \cdot \overline{L}_s)^2$ to $|Q|(\overline{N}_{fo} \cdot \overline{R})^2$, i.e. the $\overline{L}_s$ factor, which counts the average number of transitions that link two states, is replaced by $\overline{R}$, which is the average number of *ranges* that are needed to represent these transitions.

In the worst case, if a set of transitions does not include any contiguous symbols, a number of ranges equal to the number of transition labels is generated. However, as regular expressions usually handle human-readable alphanumeric characters and the ASCII code represents them with contiguous values, the vast majority of NFAs are expected to benefit from the range notation. Moreover, the advantages of this notation tend to be more visible when the NFA becomes more complex. In fact, complex NFA tend to generate more states and more transitions, hence increasing the probably to find many set of transitions that operate on contiguous values, which can be compacted with the range notation.

Finally, even with "unluckily incompressible" rule sets, it is still possible to greatly exploit the range notation starting from the second stride doubling iteration: as it will be shown in Section 7.1.2, our alphabet compression algorithm generates equivalence classes in a way to increase (if possible) the amount of contiguous labels in each transition set.

7.1.2 An improved Alphabet Compression

The state-of-the-art algorithm for alphabet compression[BC13] works well with small to medium sized NFAs but still requires prohibitive resources with multiple-stride NFAs resulting from rule sets of realistic sizes.

With respect to the state-of-the art, the algorithm presented in this section is more efficient in terms of both memory and processing requirements, hence allowing to handle larger automata, and also improves the effectiveness of the range notation.

From the memory standpoint, our algorithm (shown in Algorithm 3) requires approximately four times less memory than the one in,[BC13] as it needs just to keep an integer per symbol pair (i.e., only the transition map itself) compared to the two integers and two booleans[ii] required by[BC13][iii]. From the processing standpoint, our algorithm can potentially run at twice the speed of[BC13] as it performs a single iteration through the main data structure instead of two.

Algorithm 3 Improved alphabet compression algorithm

```
1: map[|Σ|][|Σ|] = 0
2: size = 0
3: for all q1 ∈ Q do
4:    for all q2 ∈ Q do
5:       buffer = {}
6:       for all (a, b) ∈ Σ × Σ | (q1, (a, b), q2) ∈ δ do
7:          if map[a][b] ∉ keys(buffer) then
8:             buffer = buffer ∪ {map[a][b], ++size}
9:          map[a][b] = buffer[map[a][b]]
```

[i] For the sake of precision, in case of individual symbols the Cartesian product between symbols would have generated $(122 - 96 + 200 - 199) \cdot (100 - 49) = 1377$ pairs of symbols.

[ii] The vast majority of the compilers use an integer to store a boolean value for performance reasons.

[iii] For the sake of precision, our algorithm uses also an hashed map whose number of entries is equal to the average number of equivalence classes the transitions connecting two states belong to; however this data structure can be considered negligible compared to the size of the main data structure.

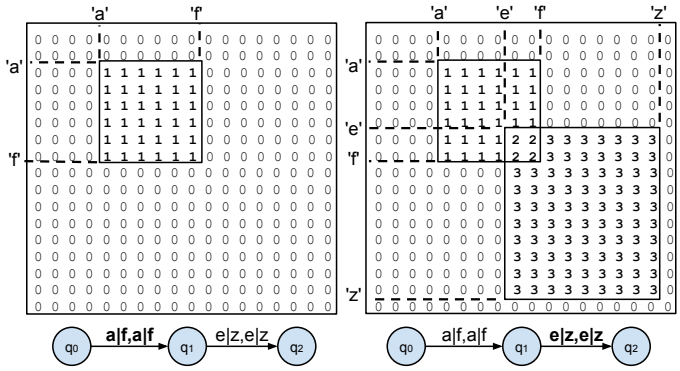

Figure 7.1: Example of the improved alphabet compression algorithm

The main idea of the new algorithm is to simplify the cluster division step (proposed in[BC13]) by using a different criterion: *all* the symbol pairs that can lead from q_1 to q_2 are remapped onto new equivalence classes, taking care of using the same new class for symbol pairs that previously belonged to the same class. For example, if the pairs (‘`aa`’), (’`bb`’), and (‘`cc`’) can lead from q_1 to q_2 and the pairs (‘`aa`’) and (‘`bb`’) were previously mapped to the equivalent class `1` while (‘`cc`’) was mapped to `0`, then (‘`aa`’) and (‘`bb`’) will be mapped to the same new symbol (e.g. `2`) and (‘`cc`’) to a new other symbol (e.g. `3`). In this way, correctness is still guaranteed, i.e. if two symbol pairs are such that from each state they can lead to the same destination states, these two symbol pairs are mapped to the same class.

The `buffer` hash-map is used to store the re-mapping operations performed at each cluster division step (it maps classes onto new classes). Initially there are no re-mapping operations (line 5). For each symbol pair that can lead from q_1 to q_2, if it was previously mapped to a class for which a re-mapping does not already exist in the buffer (line 7) a new class is created and a new re-mapping from the previous class to the new class is added to the buffer (line 8). Vice versa, if a re-mapping already exists in the buffer, this is simply used to remap the symbol pair.

Figure 7.1 shows how the algorithm works for the same FSA presented in Figure 6.3. The left-hand side of the figure shows the translation map resulting after processing the transitions from state q_0 to state q_1. All the symbol pairs that can lead from q_0 to q_1, corresponding to the square area with coordinates from ‘`a`’, ‘`a`’ to ‘`f`’, ‘`f`’, have been re-mapped to the same new class `1`. In the second step, the transitions from state q_1 to q_2 are examined. They are labeled by symbol pairs corresponding to the square area with coordinates from ‘`e`’, ‘`e`’ to ‘`z`’, ‘`z`’. This area can be divided into two parts: the former that overlaps with the area previously filled with class `1` and the latter that does not overlap with it and is still filled with the original class `0`. These two parts are re-filled with two different new classes, namely `2` and `3`, thus leading to the situation shown in the rightmost part of the figure.

Hence, as expected the algorithm has generated four different classes: class `0`, corresponding to unused symbol pairs, class `1` corresponding to the symbol pairs that can trigger only the $q_0 \rightarrow q_1$ transitions, class `3` corresponding to the symbol pairs that can trigger only the $q_1 \rightarrow q_2$ transitions, and class `2`, corresponding to the symbols that can trigger both.

The only difference between the output of the new algorithm and the output of the algorithm in[BC13] is that the new algorithm will perform a remapping even when it is not strictly necessary, thus possibly leading to non-contiguous class numbers. For example, if the algorithm generates 4 classes it is possible that these classes are numbered `0`, `2`, `3`, and `4`, as the class `1` went overwritten in some intermediate steps of the algorithm. Even if the way integer values are assigned to classes is irrelevant from the theoretical point of view, the assignment of non-contiguous values may affect negatively stride doubling (because the effectiveness of the range notation decreases). For this reason, an additional phase is added to the algorithm where the generated classes are re-numbered so as to guarantee that they are identified by contiguous integers starting from `0`. This step is performed inside

the algorithm that translates the NFA to the new alphabet (which has to be executed anyway after alphabet compression), hence adding a very small overhead to the overall process. The translation algorithm is very simple: it iterates through each transition of the NFA, and substitutes the old symbol pair with the corresponding class number. When doing so, the class is also renumbered (if needed). This adds a couple of memory accesses at each iteration and it requires to keep a dictionary to remember which classes have already been renumbered. As evident, the additional cost of class renumbering is negligible compared to the rest of the algorithm. The same applies for memory requirements, as the size of the additional dictionary is equal to the cardinality of the compressed alphabet, which is way less than the size of the support map.

Finally, as an added value, class renumbering chooses new class indexes with an heuristic that increases the amount of contiguous values in each set of transitions that connect two states: this is done in order to increase the efficiency of the range notation (i.e., to reduce $\overline{R}$). As the asymptotic time complexity of the stride doubling algorithm is proportional to $\overline{R}^2$, reducing $\overline{R}$ speeds up the next stride doubling iteration. The heuristic is based on the following idea. Let us consider the first state pair that is processed. If, for instance, the transitions that link these states are labeled by the 3 equivalence classes `563`, `236` and `1243`, the algorithm will rename class `563` to `1`, class `236` to `2` and class `1243` to `3`, thus guaranteeing the contiguousness of their symbols. For the next state pairs, only the classes that occur for the first time will be re-mapped to new (consecutive) integers. Hence, the labels of the transitions connecting the state pairs that are processed first will more likely have more contiguous symbols, while the ones of the state pairs processed last will probably have more sparse sets. By processing state pairs with more complex transition sets first, the overall number of ranges is reduced. This can be achieved in practice by sorting the state pairs according to the complexity of their transition sets before performing renumbering.

This complexity of the algorithm that generates the equivalence classes can be expressed as $|Q|^2 \cdot \overline{L_s}^2$, which stems from the fact that $\overline{L}_s$, representing the most critical factor in the formula, cannot be smaller than $\overline{R}$. Comparing the asymptotic time complexity of our alphabet compression algorithm to the one in[BC13] and reported in Section 6.3.2, we notice how the $|\Sigma|^2$ component is replaced by $\overline{L_s}^2$: this is an important improvement because usually $\overline{L_s}$ is lower than $|\Sigma|$, as typically in an NFA only few states are connected to each other with all the possible symbols of the alphabet.

7.1.2.1 Optimized remapping

When experimenting the alphabet compression algorithm with NFAs used in real network applications, we found that often different state pairs are connected by transitions with the same symbols, i.e. the symbols that can lead from q_x to q_y are the same that can lead from q_z to q_t. In this case, the remapping done by the algorithm for the first pair of states would be completely overwritten when processing the second pair of states, as they refer exactly to the same area in the map. Hence the processing of the second pair of states could be safely omitted without affecting the final result. As this case is frequent, this optimization leads to an important reduction of the processing time.

The high frequency of this case in real situations depends on several different causes. One is the already mentioned alphanumeric nature of most regular expressions. Another one is the way the NFA is typically generated from a set of regular expressions. For instance, the combination of two sub-expressions by the AND operator results in a duplication of the initial set of transitions. However, most frequently the presence of state pairs with exactly the same transition sets is caused by the '`.*`' pattern, included in many regular expressions, which causes the generation of state pairs q_x, q_y such that any symbol in the alphabet can trigger a transition from q_x to q_y. A state pair with this set of transitions forces the algorithm to update the *entire* support map, and, being these state pairs frequent, they are also responsible for a considerable portion of the time required to process the entire

NFA. For this reason, omitting the processing of transition sets that have already been processed for previous state pairs represents a simple but very effective optimization.

In order to recognize and skip state pairs whose transition sets are already been processed, the proposed algorithm exploits the above mentioned sorting of state pairs according to the complexity of their transition sets. However, in order to guarantee that state pairs with identical transition sets are adjacent, this sorting has to be refined, by introducing a secondary sorting criterion. The criterion used is not relevant, provided that a precedence relationship is defined for any two different transition sets. A simple way is to represent transition sets as strings of alphabetically ordered lists of state pair ranges (e.g. '{(`a|f,a|f`),(`a|f,i|k`)}'), and to use the alphabetic order of transition sets as the secondary sorting criterion. With state pairs sorted in this way, the algorithm compares the transition set of the state pair N to the one of the previous state pair $N-1$ and, if found to be the same, it skips the remapping step.

Using a $O(n \cdot log(n))$ sorting algorithm, the asymptotic time complexity of the sorting phase is $|Q|^2 \cdot \overline{R} \cdot log(|Q|^2 \cdot \overline{R})$, which is is negligible compared to the asymptotic time complexity of the algorithm that generates the equivalence classes.

7.2 Multi-Map Alphabet Compression

Although alphabet compression is usually a very efficient technique in terms of symbol compression ratio, it is not always sufficient to enable further stride doubling steps with the hardware resources currently available. The inability to further reduce the alphabet size mainly depends on the fact that additional equivalence classes need to be created when two transition sets share part of the symbol pairs: for instance, in Figure 6.3 *three* equivalence classes are necessary to represent the symbol pairs of *two* transition sets due to the overlap of their labels, while an equivalence class could be saved if that overlap were not present.

As it is not easy to further increase the compression ratio of the current alphabet compression technique, another way to enable further stride doubling steps is to increase the degree of parallelism of the packet processing problem. Let us denote N the number of input symbols to be processed and T the time taken to process them with a single processor. The idea is that if we cannot manage to process a *single* string of $N/2$ symbols in a time $T/2$ with a *single* processor because stride doubling is unfeasible, we may manage to process *two* different strings of length $N/2$ in $T/2$ by using *two* processors in parallel.

In order to split the matching problem into two separate problems, we partition the transition sets of the NFA into two disjoint groups in such a way that the amount of *overlapping* transition sets is minimized in each group. Then, we apply the alphabet compression algorithm separately on each group of transitions, generating two translation dictionaries, i.e. two target alphabets and two translation maps instead of a single one. This operation is likely to be feasible even if alphabet compression is unfeasible on the whole transition sets. The two dictionaries are then used to build an NFA in which the two different alphabets coexist: some transitions will be labeled by the symbols coming from the first dictionary, while the others will be labeled by the symbols of the second one, according to a criterion that will be explained afterwards. If the transition sets are partitioned into the two groups in a "smart" way, i.e. by minimizing the overlapping of transition sets, each group can be compressed more efficiently than the whole set and the overall number of generated symbols is reduced. The direct consequence of this more efficient compression is that the resulting NFA is simpler, i.e. with fewer transitions, than it would be by using the normal alphabet compression technique.

However, as the new NFA contains symbols belonging to two different alphabets at the same time, both the input translation and the matching algorithm change: input translation generates two strings, one for each dictionary, and the two resulting strings are then processed in parallel: one process considers only symbols belonging to the first alphabet while the other one considers only symbols belonging to the second alphabet, but both processes update the same set of active states. In this way, the set is updated correctly, since in each state each possible transition of the original NFA is either applied by one process or by the other one.

From a practical point of view, using more processing units to perform the matching is not a problem as the target hardware is a GPU with hundreds of available processing elements. Moreover, each processing unit will work on an NFA with in average half of the transitions per symbol, thus hopefully halving the per-symbol processing cost of each unit. At the same time, the NFA is simpler than it would be with the normal alphabet compression, which has also the effect of simplifying the next stride doubling step.

The multi-map compression algorithm can be considered a generalization of the one presented in Section 7.1.2, with the following modifications: (i) During alphabet compression, two maps are used. For each pair of states q_1, q_2, the algorithm updates only one map, selecting the one on which compression is better (trivially, the one on which the update causes the generation of less new equivalence classes). (ii) Finally, when all the state pairs have been processed, and the equivalence classes have been separately renumbered in each map, the new NFA is built by using both translation maps: on average, half of the transition labels will be replaced by equivalence classes belonging to the first map, while the other half will use symbols belonging to the second map.

Due to the NP-Completeness of the problem, the proposed 2-map algorithm cannot find the optimal distribution of symbols but the greedy heuristic it uses enables finding fairly good solutions without increasing so much the time complexity of alphabet compression with respect to the original algorithm: the required time and memory simply double, because two maps must be used and updated in place of one. However, thanks to the better compression provided, subsequent stride doubling steps will be performed faster and with less memory requirements with respect to the original technique.

In literature, several alternative heuristics are available: for instance, we considered GRASP (Greedy Randomized Adaptive Search Procedure) algorithms, Hill-climbing variants and the top-down algorithm proposed in,[KSE08] but none of them allowed us to outperform our approach in terms of required time and compression efficiency with the rule sets at our disposal.

7.2.0.2 Run-time packet processing

Even if, in theory, multi-map alphabet compression should be able to greatly improve the maximum achievable stride level and the processing throughput, performing further stride doubling and alphabet compression iterations to an NFA that has been compressed with the multi-map algorithm causes additional issues that must be properly handled, and that can be explained by taking into account the example in Figure 7.2. When a 2-map alphabet compression is performed on the uncompressed 2-stride NFA of Figure 7.2a, the transitionfrom q_1 to q_2 and those from q_2 to q_4 are assigned to the first support map, while the transition from q_2 to q_3 is assigned to the second support map. The support maps generated in this way are shown in Figure 7.2e, while the compressed NFA is shown in Figure 7.2b.

If the input string is '`a,b,c,d`', the following two strings are generated by applying the two translation dictionaries: `1,2` (by using the first map), and `0,4` (by using the second map). The packet processor has to concurrently process symbols `1` and `0` at the first iteration, and symbols `2` and `4` at the next one. At the first iteration, if the active state set includes only state q_1, the transition from q_1 to q_2 is triggered by symbol `1`, thus adding q_2 to the new active state set. Then, at the next iteration, the

active state set includes only q_2 and both the transitions from q_2 to q_3 and from q_2 to q_4 are triggered by symbols `4` and `2` respectively. Thus, the final active state set includes exactly states q_3 and q_4, as expected.

If now a new stride doubling step is performed on the NFA, getting the 4-stride NFA shown in Figure 7.2c, the presence of two translation maps complicates the input string translation and the matching algorithm.

In order to realize why, let us see what happens if we apply again the same algorithms explained above. The input string '`a,b,c,d`' is translated into the two strings `1,2` and `0,4`, using the two maps. Then, the matching algorithm consumes the pairs of symbols `1,2` and `0,4` concurrently. If initially the only active state is q_1, the pair `1,2` triggers the transition from q_1 to q_4, while the pair `0,4` does not trigger any transition. Thus, the final active state set includes only state q_4, which is wrong.

The reason why the result is wrong is that the `1,4` transition has not been fired while it should have been. The particularity of the symbol pair `1,4` is that its two components belong to different translation maps while our translation algorithm only generates two translations: one made only of symbols of the first alphabet and the other one made only of symbols of the second alphabet. In order to make sure that we get the right result of matching at stride level 4, it is then necessary to generate more translations of the input string, including all the possible combinations of the two translation maps. In our example this implies generating 4 translations, i.e. the strings `1,2`, `1,4`, `0,2` and `0,4`, where the first string is obtained by using the first map for both symbols in each pair, the second one is obtained by using the first map for the first symbol and the second map for the second symbol in each pair, and so on.

If now we apply again the 2-map alphabet compression algorithm on the 4-stride NFA, we get two new translation maps, shown in Figure 7.2f, and the new compressed NFA shown in Figure 7.2d. Of course, having introduced two new alphabet translations, the 4 input strings we had for the uncompressed 4-stride NFA have to be translated into 8 strings: 4 obtained by applying the first new translation map and 4 obtained by applying the second one. Taking again the above example, the string `1,4` is translated by the leftmost map of Figure 7.2f into `y` while the other 3 strings are all translated into `0`. Using the rightmost map, the same four strings are translated into `z` (generated from string `1,2`), `0`, `0`, `0`.

In general, with the proposed approach, at each alphabet compression step the number of strings to be processed in parallel is multiplied by the number of maps (with 2 maps it is doubled). At each stride doubling step, instead, the number of strings to be processed in parallel is squared, and the length of the input strings is halved.

Then, after n combined stride doubling and alphabet compression steps, the number of strings that the matching algorithm must process in parallel becomes $S_n = m^{2^n-1}$ where m is the number of maps used for alphabet compression, and the length of the input strings becomes N/n, where N is the original length of the input string.

The increase in the number of input strings to be processed in parallel is not necessarily a problem: it just makes the processing operation a more parallel task, and even if we are "wasting" some threads, the GPU application can be made smart enough to minimize this overhead by greatly reducing the total amount of memory accesses performed by iNFAnt (which is the actual main bottleneck of iNFAnt). This can be achieved by exploiting the fact that some of the symbols occurring in the string translations are redundant. Specifically, multiple occurrences of the same symbol in different strings at the same offset are redundant because they will produce the same target states. Detecting redundant symbols and avoiding their processing is a way to reduce memory accesses.

Redundancies happen frequently and they can be observed even in the simple example that has been presented so far: of the 8 1-symbol strings generated from the translation of the original 4-symbol input, 6 strings are identical (`0`). Hence, five of of them could be be ignored as they would bring no additional contribution.

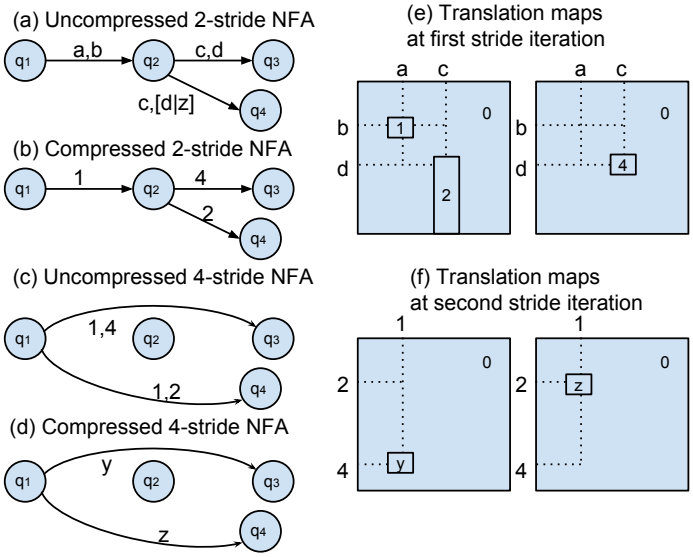

Figure 7.2: Multi-map compression of a sample NFA at 2x and 4x stride levels

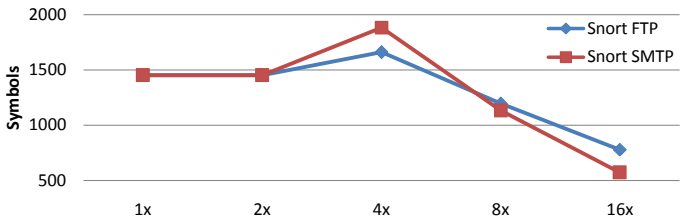

Figure 7.3: Amount of non-repeated symbols present in random input data streams when translated to be processed with 2-map multistrided NFAs

Section 7.2.0.3 analyses symbol redundancy and its impact, while Section 7.3 explains how iNFAnt has been optimized in order to recognize and avoid redundancies without introducing too much overhead.

7.2.0.3 Redundancy in input string translations

Experimentally, it is easy to show that the phenomenon of symbol redundancy occurs with any rule set, and that, after a certain stride level, redundant symbols tend to increase. The experiment consists of generating the NFAs at different stride levels for the considered rule set, using the stride doubling and the 2-map alphabet compression algorithms proposed here. Then, various pseudo-random input packets are generated, and, for each stride level they are translated by using the dictionaries originated by the alphabet compressions, as previously explained. Then, the input strings so generated for each stride level are reduced by removing redundant symbols (i.e. equal to other symbols occurring in other strings at the same position). For example, if we have the 2 strings `1,2,3,4`, and `1,0,3,3`, the symbols in bold are removed because they are repetitions of symbols occurring at the same position in the other string. In this way, only the meaningful symbols remain. Finally, the total length of all the reduced strings, which represents the total number of non-redundant symbols, is computed.

Figure 7.3 plots the results of this experiment using 1500 bytes long packets and the rule sets used by the popular Snort NIDS to analyze FTP and SMTP packets (each plot refers to a single rule set). These rule sets have been chosen because, thanks to their reduced size (about 20 simple regular

expressions), they allow to reach high stride levels. More details about these rule sets are presented in Chapter 8.

In absence of symbol duplications there would be no length change when moving from 1x to 2x (because 2 strings of length $N/2$ each are generated), but there would be a length increase of two times when moving from 2x to 4x (because 8 strings of length $N/4$ each are generated), an increase of 16 times when moving from 4x to 8x (because 128 strings of length $N/8$ each are generated) and so on. In Figure 7.3, instead, we can see that from 2x to 4x the data increase is way less than the doubling that would occur in the absence of symbol duplications, while at further stride levels there is even a linear reduction of the total number of symbols rather than the exponential growth that would occur in the absence of symbol duplications.

An intuitive explanation of this phenomenon follows. As the stride level increases, it becomes more and more difficult to find "overlapping" transition sets that need to be processed separately in order to avoid generating extra equivalence classes: this is because at every stride level transition sets are compressed into fewer single classes and, at the same time, the size of the support maps increases. For this reason, support maps become increasingly less populated (particularly the secondary ones), and the amount of input string patterns with several different possible translations dramatically decreases at each stride level.

This trend can also be understood considering the asymptotic behavior: in the hypothesis we would be able to compute the 2^k-stride NFA for arbitrarily large values of k using 2-map compression, we would reach a limit situation where the input string is translated into strings of length 1. In this extreme case, if N is the number of acceptance states (which typically corresponds to the number of regular expressions in the rule set), we would necessarily end up with $N+1$ meaningful symbols, each one leading from the initial state to one of the acceptance states, plus an extra symbol representing all the other cases. Hence, asymptotically, the number of meaningful symbols tends to $N+1$, where N is the number of acceptance states of the NFA.

Consequently, as the rule sets used in the experiments presented in Figure 7.3 are composed of about 20 regular expressions each, it can be expected that the number of meaningful symbols to be processed will tend to about 20, as well as the alphabet size. This explains the reduction that we can already observe starting from the 8-stride NFA (on which 128 different strings are generated when translating each input string).

The behavior in the "transient phase", represented by the first stride levels, as well as the maximum number of useful symbols, depends on the complexity of the rule set taken in consideration. However, in our test cases, all the rule sets for which it was possible to go beyond the "4x barrier" have demonstrated to generate a huge amount of redundant, useless symbols.

Based on these considerations, iNFAnt has been optimized so as to make it able to exploit this phenomenon and avoid the processing of useless symbols, so that a good performance can be achieved at any stride level, as shown in the next section.

Finally, it is worth noting that the maximum number of useful symbols to be processed does not depend on the input string: there is, in fact, an "upper bound" that depends only on the rule sets, thus making it impossible to forge a malicious incompressible input data string.

7.3 Refining the iNFAnt architecture

This section presents the adaptation of iNFAnt to the new algorithms proposed in the previous sections.

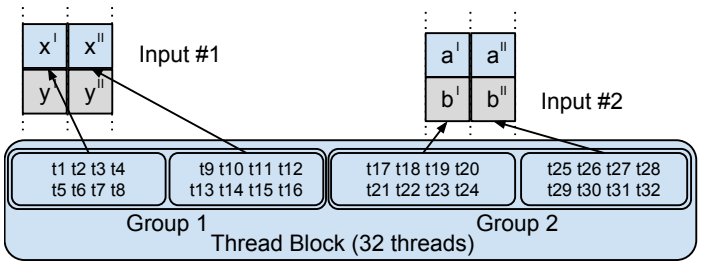

Figure 7.4: Thread task scheduling during the processing of 2 input messages translated in 2 different strings each

nVidia GPUs are architecturally composed of execution units, each one capable of running one or more thread blocks in parallel, with 32 to 1024 threads per block.

The original version of iNFAnt followed this architecture closely, by assigning a different input string (i.e., a network packet) to each block and by using one thread per transition in the computation of the next active states. When operating at high stride levels this choice becomes no longer adequate because the average number of transitions per symbol tends to decrease. With less than 32 transitions per symbol, the architectural constraint of having a minimum of 32 threads per block implies that some threads remain idle, thus wasting resources.

For this reason, the thread hierarchy has been changed in iNFAnt so that now the "thread group" concept is no longer coincident with the concept of hardware thread block: each thread block can now include several thread groups, each one assigned to a different input string.

This re-organization of threads not only enables a reduction of the number of inactive threads, but also helps to better manage the parallel processing of the several string translations that originate from multi-map compression. In the new algorithm, the threads of each group cooperatively process all the translations of a single input string. An example where a block of 32 threads is used to process two input strings is shown in Figure 7.4. The picture refers to a 2-stride NFA compressed with two maps, so that two translations of each input string have to be processed in parallel. The upper part of the figure shows part of the two input strings, where each column represents a different translation. For example, in `input # 1`, x' is the current symbol of the first translation while x'' is the current symbol of the second one. The thread block is divided into two groups, each one assigned to a different input string, and the threads in each group are divided among the two translations. In this way, the parallel processing of the transitions associated to each input symbol is kept (in the example, each symbol is processed by eight threads), as in the original iNFAnt version.

Thanks to the nVidia GPU hardware architecture, processing different strings in the same thread block also enables a reduction of memory accesses. In fact, as discussed in Section 7.2, the probability that two or more threads have to process the same symbol at the same time in the same thread block is very high, due to the symbol redundancy phenomenon discussed in Section 7.2.0.3. When this happens, the nVidia Memory Management Unit (MMU) can efficiently join identical memory requests to be forwarded to main memory, thus efficiently reducing the real amount of memory accesses. Thanks to this feature, as far as we manage to process all the translations of an input string by threads belonging to the same block, the amount of really performed memory accesses corresponds to the numbers plotted in Figure 7.3. This means that even if the number of symbols that must be processed increases at each stride doubling, the code could still increase its performance by ignoring this overhead.

However, this is possible as far as the number of translations is small enough to fit into a single block. In order to make this possible in a broader range of cases, an additional optimization has been added to the code that performs input translation: the data translator "compresses" its output by completely dropping the strings that contain only redundant symbols. This trivial compression mechanism dramatically reduces the number of translations that must be processed for each input

string. This was confirmed by our experiments: even at the highest stride levels and with the most complex rule sets we could handle, it has never been necessary to process more than 16 translations per input string.

This reduction of the number of translations also reduces the total number of threads required, which contributes to increase the scalability of the solution, because the maximum number of threads is an hardware constraint. Moreover, if the number of strings to process increases too much it is even possible that the data transfer between the CPU and the GPU becomes the main bottleneck of the system, thus limiting the maximum achievable throughput.

8

Performance Measurement Results

How fast are the new algorithms? How to compare them with state of the art?

Contents

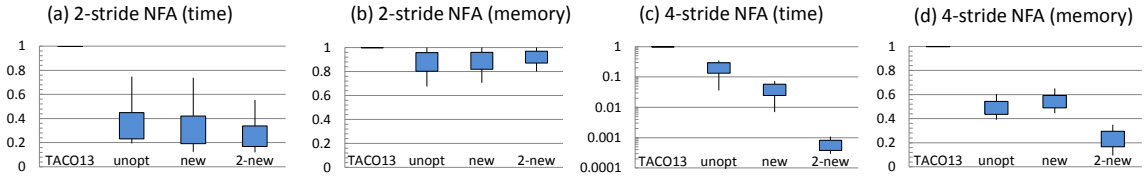

Figure 8.1: Required times and maximum required memory comparison between four stride doubling tool sets when generating 2 and 4-stride NFAs

The experiments described in this chapter aim at evaluating the performance of the algorithms that have been just proposed, both those for building multi-stride NFAs, and the new GPU-based iNFAnt matching algorithm optimized for dealing with multi-stride NFAs.

The platform used to run all the tests is a PC with an Intel i7-960 quad core CPU running at 3.20Ghz, 12GB of DDR3 as main memory and an nVidia Tesla c2050 as GPU board.

Table 8.1: The rule sets used for testing

Name	Rules	Size (KB)	States	TpS	Throughput
Small					
Snort FTP	17	4	132	15	951,55 Mbps
Snort SMTP	26	12	433	30	809,48 Mbps
Medium					
Snort 534	534	208	9.538	127	289,89 Mbps
Snort HTTP	189	92	3.538	295	273,71 Mbps
Large					
ET HTTP	457	428	18.425	525	500,68 Mbps
Snort Full	1514	1100	47.168	1.696	25,62 Mbps

Table 8.1 lists all the rule sets used in our tests along with their size, TpS, and average throughput as obtained by iNFAnt without multi-striding. The rule sets have been classified into three categories, depending on the complexity of the generated NFA (small, medium and large). Some of these rule sets have been extracted from the rules of Snort and of EmergingThreats(ET) [i], which is another commercial IDS. Snort Full includes the full Snort rule set, with the only exclusion of the rules that have no standard PERL syntax. Snort 534 is a selection of 534 rules used as benchmark in.[BC07] The other Snort and ET rule sets have been built by selecting only the rules of a single protocol, specified in the name of each rule set.

8.1 Results (multi-stride NFA generation)

Let us denote “new” the new algorithms for building multi-stride NFAs, and “2-new” the version with two maps for compression. The performance of these algorithms has been measured and compared with the one achieved by the previous state of the art algorithms by Becchi and Crowley.[BC13] The latter are denoted “TACO13”, which is the acronym of the conference where they have been presented. As no public implementation of these algorithms was available, an ad-hoc implementation of the pseudo-code presented in[BC13] has been developed for our purposes. Other algorithms presented in literature, like,[BTC06, YKGS11, CRRS10] are so slow that they cannot even complete all the benchmarks we used in reasonable time. For this reason they have not been considered relevant for our analysis. In order to evaluate the contribution of the state pair sorting and redundant string elimination optimizations, the results are also compared with those obtained with a version of our multi-map algorithms that does not include these optimizations (denoted “unopt”).

[i] `www.emergingthreats.net`

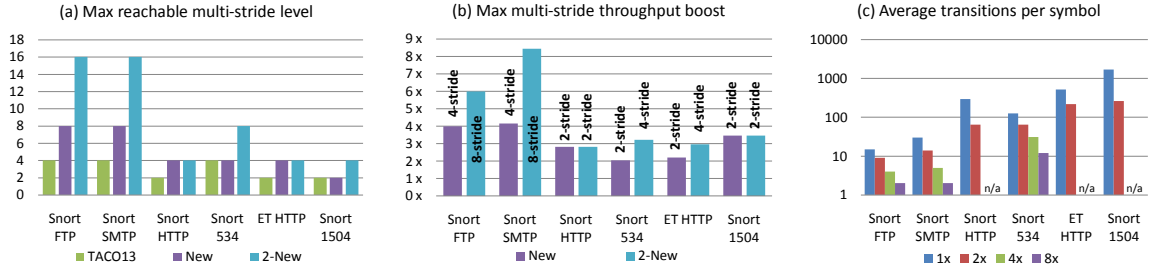

Figure 8.2: Detailed performance of every multi-strided rule set in terms of: (a) maximum stride level reachable in a 24-hour window, (b) maximum throughput boost and (c) average amout of transitions per symbol

The performance of building the multi-stride NFA has been evaluated by measuring the total time and the maximum amount of memory taken by each algorithm for building the multi-stride NFAs of all the considered rule sets. The time taken for 2-stride and 4-stride NFAs have been measured separately. Measurements have been repeated several times and average, standard deviation, min and max have been calculated. The results, plotted in Figure 8.1, have been normalized taking as reference the performance of the "TACO13" algorithm. Vertical lines represent the minimum and maximum obtained values while rectangles represent the average value plus and minus standard deviation.

At stride level 2, especially for what concerns memory requirements (Figure 8.1b), the various algorithms behave nearly the same because the most critical memory structure (the support map of size $|\Sigma|^2$) is still very small and even smaller than the memory required to load the libraries used by the algorithm.

Things change by increasing the stride level from 2 to 4, as it is possible to see in Figure 8.1c,d. At stride level 4, the "unopt" algorithm is, on average, 10 times faster than "TACO13". Adding state pair sorting and redundant string elimination further reduces the time taken by another order of magnitude, while with 2-map compression the time taken is more than three orders of magnitude lower than that of "TACO13". Memory consumption measurements are close to the theoretically expected values: the new algorithm uses a single support map of integers in place of the four support maps of the "TACO13" algorithm and thus it requires less memory, about an half (because even at 4x there are still additional data structures using a non negligible portion of memory, especially with simpler NFAs), while multi-map alphabet compression greatly reduces the alphabet size, thus reducing the size of support structures as well.

Figure 8.2a shows the results of another experiment that has been performed in order to evaluate the maximum stride level that can be obtained in a 24-hours time span with the different algorithms and rule sets. For a given algorithm and rule set, the experiment consists of applying the algorithm to iteratively double the stride level of the NFA until the 24 hours limit is reached. The maximum obtained stride level is reported in the graph. The "new" algorithm outperforms "TACO13" in four cases out of six, while "2-new" does even better because it iterates at least one time more than "TACO13" with all the rule sets, and even more times in the simplest cases.

One of the motivations for the processing throughput increase expected when using multi-map alphabet compression is the expected reduction of the average amount of transitions per symbol when the stride level increases. This reduction has been experimentally observed on the selected rule sets, as shown in Figure 8.2c which reports this figure at various stride levels for various rule sets. This value is directly related to the average cost of processing a symbol given by iNFAnt, and as described in Section 7.2 it is expected to be at least halved at every iteration of the multi-striding algorithm. As it is possible to see in Figure 8.2c, this is always true at every stride level and with any rule set. Moreover, with very large rule sets this effect seems to be amplified. For example, with Snort 1504 the reduction rate is higher than 80%.

8.2 Results (data processing)

This section presents the experimental evaluation of the throughput boost made possible by the new iNFAnt algorithm and by multi-striding. The throughput has been measured, for each rule set, using different multi-stride NFAs (the ones obtained using the different generation algorithms compared in the previous section), and using a workload made up of real traffic captures taken from our University network.

The measured throughputs have been normalized with respect to the throughput obtained by using iNFAnt with the 1-stride NFA for the same rule set and the resulting speedup values are reported in Figure 8.2b.

In order to correctly interpret this graph it is important to note that the "new" algorithm produces the same NFA also produced by "TACO13" and by the tools described in,[CRRS10] and with these NFAs the speedup of each stride doubling is always about 2x, as foreseen in.[CRRS10] For this reason the speedup values reported in this graph are aligned with the values in Figure 8.2a, apart from few "lucky" cases in which the multi-strided NFA proves to be faster; the only cases in which the maximum throughput boost is reduced, with respect to the maximum achievable stride level, are when the generated NFA is so big that it does not fit in the memory of the GPU, and for this reason we have been forced to report the throughput value obtained by using the NFA of a lower stride level. This happens with several rule sets, especially at high mutistride levels. For what concerns NFAs generated with the "2-new" algorithms, instead, the throughput boost at each stride doubling is not always equal to 2x but it changes depending on the rule set, and it is usually slightly lower than 2x. However the graph shows that the overall speedup is nearly always greater than the values achievable by using the other techniques. The only two rule sets for which this does not happen are "SnortHTTP" and "Snort1504". Once again this is due to the memory limits of the GPU which prevent the largest NFAs from being loaded.

It is important to remark that the memory limits we encountered are an hardware issue that could be overcome by using more recent hardware. The actual trend of GPGPU vendors is to constantly increase the amount of memory of their video boards: for this reason it is likely that in the near future the current hardware limits will be overcome, and the usefulness of the "2-new" algorithms will be increased.

8.3 Efficiency of multi-map alphabet compression

As the multi-map alphabet compression technique produces NFAs that are completely different from those of the other techniques, additional tests have been performed with randomly generated rule sets, in order to show that the benefits of multi-map do not depend on the rule sets that have been chosen, but they apply to any rule set. The random rule sets have been generated using a generator that implements the generation strategy described in,[BFC08] but extends it by also including the possibility to generate "nested" regular expressions, like $ab(c.*d)+$, thus yielding more realistic rule sets. Moreover, this generator lets one specify the desired value of ρ, that is the percentage of wildcards that occur in each regular expression. With $\rho = 0$, just plain sequences of characters (or sets of characters) are generated. When $\rho = 0.5$, instead, each generated character (or set of characters) is associated to one of the possible wildcards, like repetition operators such as $+$, $*$, $\{a, b\}$ or the optionality operator $?$.

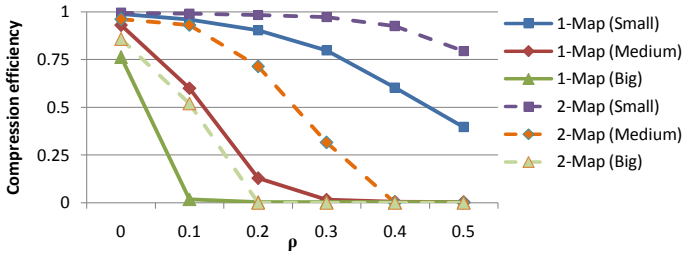

Figure 8.3: Comparison between normal and multi-map alphabet compression in terms of compression efficiency

Three different classes of rule sets have been randomly generated: "small" rule sets, composed of 20 regular expressions, each one 20 characters long, "medium" rule sets, composed of 50 regexps each one 50 characters long and "large" rule sets composed of 100 regexps each one 100 characters long. For each class, rule sets with different values of ρ have been generated, in order to study how this parameter affects the results.

For each generated rule set, the 2-stride NFA has been built by using both "new" and "2-new". Then, for each NFA a "compression efficiency ratio" has been calculated as $ratio = (|\Sigma| - |\Sigma'|)/|\Sigma|$, where $|\Sigma|$ and $|\Sigma|'$ are the alphabet sizes of the NFA measured before and after compression (`1` means that the alphabet size has been reduced to a negligible value, while `0` means that the compressor did not reduce the alphabet size at all). Figure 8.3 shows the obtained results. For each value of ρ, 10 rule sets have been generated for each class and the average compression efficiency has been plotted in the figure.

The plot shows that the efficiency of the new algorithms (dashed lines) is always better than that of the original algorithms (continuous lines). Moreover, it is possible to notice that with medium or large rule sets the efficiency drops to 0 when ρ becomes too large (i.e. rules become too complex). Multi-map compression pushes forward this limit.

8.4 Input translation overhead

The last experiments aim at dispelling any concern regarding the possible overhead required to translate the input data before sending them to the packet processor. The complexity of this task, in fact, increases with the stride level, especially with multi-map compression. However, luckily, there are techniques to hide this overhead. Specifically, this operation can be performed by the CPU as a pre-processing task that is pipelined with the GPU processing task.

By measuring both the translation throughput and the GPU processing throughput it is possible to compare them so as to determine which task is the "bottleneck" of the system.

Figure 8.4 shows throughput measures taken with our fastest rule set, "Snort FTP". As it is possible to notice here the only case in which the translation throughput becomes the bottleneck of the system is with the "2-new" algorithm, at the highest stride level. In any other case the overhead due to input translation can be safely ignored as it is negligible.

It is important to remark that the throughput of the translation algorithm strongly depends on the amount of performed memory accesses. Since the amount of memory accesses is only affected by the number of used dictionaries and not by their size, this means that the translation does not change depending on the NFA but it only changes depending on the stride level. This implies that, for any "slower" rule set, the processing throughput values will be reduced while the translation overhead remains unchanged. Moreover, the current symbol translation code could be greatly improved:

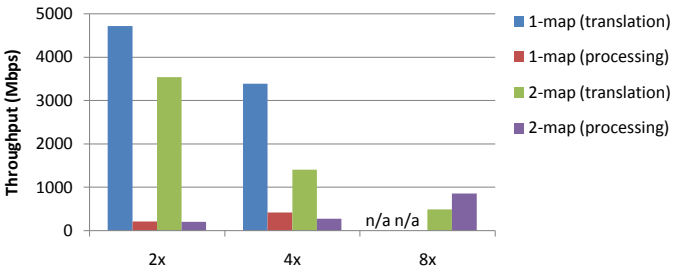

Figure 8.4: Data translation throughput versus data processing throughput

features like the data compression presented in 7.3 may be exploited to reduce the global amount of memory accesses, and thus the global translation throughput.

In conclusion, the translation throughput may become an important problem to face in the future, when the maximum achievable stride level and thus the final processing throughput will further increase, but currently it does not represent a real problem, as the throughput of large rule sets is still not so high, even at the highest reachable stride levels.

9

The DELTA Framework

An alternative approach to data analysis: what happens when all the devices under analysis are also involved in the processing of the same data they produce themselves?

Contents

9.1 Principles

Another way to cope with the discussed performance limits is the parallelization of the tasks: in particular, data centers may possibly represent the best choice from the performance point of view, due to their intrinsic scalability and the efficiency of its computation resources. However, remotely performing data anlysis does present some critical issues.

The first problem relates to the network bandwidth needed to duplicate and redirect (part of) the traffic from capturing probes to the data center, which results in increased (possibly doubled) bandwidth requirements across the network and excessive amount of traffic concentrated in the vicinity of the data center.

Secondly, traffic is usually captured on network backbones where flows are aggregated from a large number of sources and can be captured by a single probe, that is an expensive piece of equipment. Some information normally available closer to edge of the network (i.e., near end hosts) might not be accessible on the backbone. For instance, the capability to associate each packet to the originating device (or user) may be impaired if traffic is captured after it has passed through a Network Address Translator (NAT) that substitutes the source address corresponding to several different devices with a same address. Also, associating a communication session to the application generating it is possible in the most accurate and reliable way only if a probe is embedded in the host stack, as proposed in.[GSD+09]

Finally, centralized execution of analytics can be problematic when the network owner would like to contract it to a different organization, but sensitive information cannot leave the network, while it is acceptable to share specific metadata or aggregated information on which analytics could be based.

Distributing the execution of the analytics through the network would provide a solution to all of the above problems. In such a scenario some functions (e.g., device/user identification) can be performed at the edge of the network or even in the hosts themselves. Others can be executed within specific network boundaries satisfying administrative constraints (e.g., within the corporate network where sensitive information can be safely propagated). Finally, processing components that require cross-correlation of data from different sources (e.g., botnet detection) or large amount of processing and storage capacity can benefit from execution in a centralized, resource-rich environment, such as a data center. In the above architecture, information can be filtered and aggregated at each processing point of the network, hence reducing the additional traffic destined toward the datacenter. On the other hand, since network analytics require sophisticated algorithms implemented by large and complex software, it is key to be able to distribute through the network the execution of existing code without rewriting it from scratch or even just heavily modify it. Moreover, developers should not have to deal with the additional complexity of distributing functionalities, coordinating their execution, and moving data among the various components.

This chapter proposes a framework and methodology to transform centralized network analytics into distributed ones, where both data capture and part of the processing can be performed at any network node, possibly at the edge of the network. The Dynamically Evolving Lightweight Task Allocation (DELTA) Framework is implemented to demonstrate the feasibility of the approach by deploying it to distribute the analytics of a preexisting traffic processing system called MOSAIC.[XSL+13] Specifically, this work includes four contributions:

- A methodology for splitting complex analytics into a chain of modules, where a set of requirements can be specified for each module to properly operate.

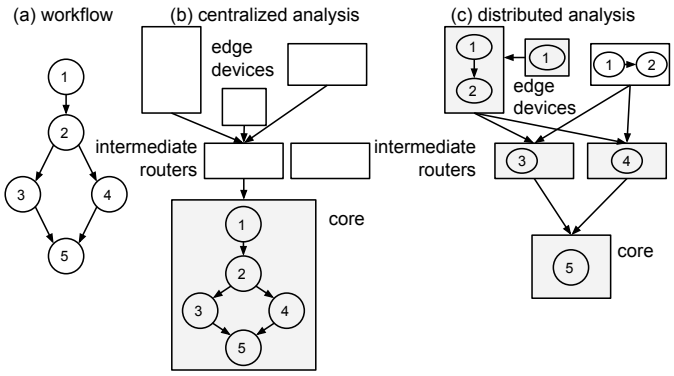

Figure 9.1: (a) example of data analysis workflow, (b) centralized task scheduling solution and (c) distributed task scheduling solution

- A framework for declaring the resources available in each network element and possibly constraints for their use (e.g., no noticeable performance degradation on the hosting systems, or processing limited to a certain type of data, such as non sensitive one).
- An algorithm to distribute the identified modules throughout the network based on (*i*) a cost function, (*ii*) the set of constraints associated to each module, and (*iii*) the resources and constraints declared for each network node.
- A testbed in which the different modules composing a traffic analysis system are automatically allocated, loaded, executed and dynamically migrated from device to device to avoid overload, thus demonstrating the feasibility of the proposed approach. The module (re)allocation is performed by scalable, distributed algorithms and is completely transparent to the developer, who can focus on writing the code that implements the algorithms.

Finally, we demonstrate the effectiveness of this solution by transforming the (previously monolithic) MOSAIC$^{\text{XSL}^+13}$ data analysis framework into a distributed system. The main expected benefits is a notable reduction in the amount of data sent from the network under analysis to the centralized analyzer. For instance, as it will be detailed in Section 10.2, in our case only an average stream as little as 10.8 Kbps was needed to transport the valuable information extracted from about 24 Mbps of continuous network traffic when using a distributed version of MOSAIC. Furthermore, this huge compression rate of the information guaranteed by the distributed analytics makes the network overhead due to the usage of a remote analyzer completely negligible.

9.2 Design of DELTA

At a high level, our proposed approach aims to split a monolithic application running in a datacenter into a set of tasks executed across the network (possibly close to its edge), such that they either add extra information to the traffic and/or process them to prevent the voluminous raw traffic to be sent to the centralized analyzer.

Consider the sample workflow in Figure 9.1 (a) which depicts the processing flow of an hypothetical data analytic as a set of interconnected modules. Let Task 1 be a simple filter that inspects protocols from raw traffic captures and filters out protocols that are not of our interests (e.g. encrypted or proprietary protocols); Task 2 be an aggregator that groups the packets belonging to the same source (device) into a single flow, while Tasks 3 and 4 be data extractors for two different types of information from the flows, such as user ID extraction on one side and traffic type identification on the other. Finally, Task 5 collects all the flows and evaluates traffic latency on a per-device, per-protocol basis.

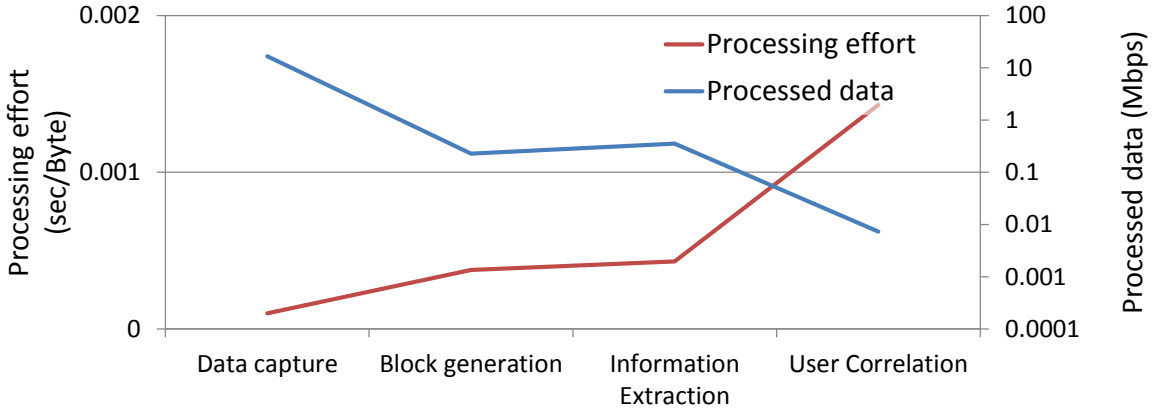

Figure 9.2: Amount of data each MOSAIC task has to process versus the corresponding required computing effort

Among all the tasks, Task 5 requires the majority of computing power because it is the task that runs data mining algorithms collected from many end-hosts from across the entire network. This is a common trend across data analysis algorithms: the first tasks are usually fast as their main purpose is to reduce the volume of data to process, while the last ones perform the most complex operation over the entire data sets. A practical example of this phenomenon is shown in Figure 9.2: in this case the MOSAIC tasks are taken as a reference, and it is possible to notice how the User Correlation task (the last one in the workflow) is the most expensive in terms of time required to process its input. More details about this workflow will be given in Section 10.

To separate the tasks and dynamically allocate them to different devices, the following four main components are needed: (i) a methodology, applicable to the main part of data analysis algorithms, to split the application in functional blocks, each of them with their own requirements; (ii) a way to describe the resources available in the network along with their policies; (iii) the definition of a cost function that has to be minimized; and (iv) a scheduling algorithm that allocates tasks to devices based on the resource availability, the cost, and the policies. The following subsections will describe the four components in details. Table 9.1 lists the main symbols used in the following, which will be explained in the subsequent sections in detail.

9.2.1 Splitting algorithms in sub-tasks

This subsection describes how we split a monolithic analysis tool into a series of sub-tasks, followed by our method on defining computational resources required for each task. In the previously described example of Figure 9.1, the traffic analysis algorithm was defined as a series of interconnected tasks. This concept can be extended to a very wide range of data analysis algorithms: as an analytic has to face huge amount of data, in fact, they typically have to be structured as stream processing tasks able to work in (almost) real time, because storing these enormous volumes of traffic before processing them might be unfeasible. For this reason, we use Directed Acyclic Graphs (DAG, [THW02]) to model traffic analysis workflow, as other stream processing algorithms do.

As structuring data analysis algorithms as a pipeline of building blocks is required by the nature of the data they have to face, we propose a methodology that exploits this common behavior by asking the developer of the analytic to just define each task and interconnections among them, while how these tasks will be distributed across the network and how data will be moved is up to the underlying framework. However, as the developer might still need to control some of the distribution aspects, the proposed methodology also require to define “task manifest” files, one per each processing task,

Table 9.1: Conventional symbols and notation used throughout the text

Network load functions and components	
$L(n)$	Function calculating a global load score given a network n
$F(x)$	Function calculating a device load score given a configuration x
$f_*(x)$	Main components of the $f(x)$ function, where $*$ can be cpu, mem or net
α, β, γ	Arbitrarily chosen coefficients used to assign priority between cpu, mem and net components of $f(x)$
$f_c(x)$	Additional component added to the $f(x)$. Used to apply custom additional policies
Indicators about device capabilities	
cpu_{ref}	Maximum processing bandwidth of the reference task ref
mem_{avail}	Available memory of a device
bw_n	Nominal available bandwidth for a network interface n
Coefficients to calculate tasks load	
bw_i	Bandwidth of the data stream directed to task i
cpu_i	Processing time of task i normalized over its incoming bandwidth (bw_i) and over the processing time of the reference task (bw_{ref})
mem_i	Amount of statically allocated memory for task i
$mem_{\Delta i}$	Amount of dynamically allocated memory for task i depending on its incoming bandwidth bw_i
io_i	Input/Output ratio of task i (used to automatically estimate bw_i of all the tasks just by knowing the bandwidth of packet capture units deployed in the network)

specifying possible custom policies. These policies might limit the mobility of a task to a restricted range of devices (e.g., some devices may not be trusted), might forbid to create multiple instances of a task or might be used to extend the DAG model by adding new bi-directional communication channels across tasks, often needed to exchange control information.

Task mobility, in particular, is limited by using a numerical parameter called "scope", used to categorize devices: a network operator must define a *scope* value for each device, while for each task $scope_{min}$ and $scope_{max}$ values determine lower and upper bounds allowed for this task. For instance, tasks can only be moved across devices whose their *scope* falls within the range defined by the tasks themselves.

It is up to the analytics developer to choose a criteria to assign scopes. To name few examples, it could be possible to use them to identify devices with particular hardware capabilities, for instance to avoid putting memory-hungry tasks in devices with a small main memory, or choosing only devices equipped with a Graphical Processing Unit (GPU) chip. Another possible use is to define define topologycal areas: in this way a company particularly sensible to privacy issues might use scopes to force anonymizing tasks to be performed before the processed data leaves the local network.

The task manifest file also contains information about the resources required to run each task. Three types of resources are considered: CPU, main memory and network bandwidth. For what concerns CPU usage the processing speed of each task is compared to a reference algorithm and the speed ratio between the two is used as a parameter: for instance, if the task i is two times slower than the reference algorithm, its cpu_i score will be 2. Memory usage instead uses two parameters: a fixed value mem_i, representing the fixed amount of memory the task allocates when a new instance of it is created, and a variable value $mem_{\Delta i}$ representing how much additional memory is needed depending on the amount of data the task takes in. Given the amount of data the task is processing at a given time, defined as bw_i, we can calculate the total amount of memory used by the task in that instant with the formula $mem_i + bw_i \cdot mem_{\Delta i}$. Finally, the task manifest defines a coefficient io_i representing

the reduction/expansion[i] ratio between the output data stream and the input data stream, in terms of bytes per second. As the amount of data captured by the probes is known, the io_i is used to calculate the bw_i of all the tasks in a network.

9.2.2 Defining the available resources

Having defined resources required for the tasks, we present now how we define the resources available on the different devices. Similarly to each task, a device on the network has "device manifest" files with the following information: the aforementioned *scope* parameter, the measure of maximum CPU throughput cpu_{ref} with respect to a reference algorithm, and the available memory mem_{avail}. Finally, the bandwidth available per each network interface bw_n available on the device also needs to be specified.

9.2.3 Computing the cost function

Formally, the problem we aim to solve regards assigning, to each device, a series of tasks according to resource occupation constraints (it is never possible to use more resources than the amount advertised by the devices themselves) and to other custom constraints, related to other possible necessities of the tasks. An example of this mapping can be seen in Figure 9.1(c). In this case we can assume that the device in which only Task 1 is executed has few resources, so it is "helped" by another device to perform the first two steps of the analysis. Then, as Task 3 and 4 might have different requirements, they have been placed in two different routers, each of them particularly well suited to run one of the tasks. Finally, everything has to go to the core, where the information crossing occurs.

If both resource requirements and availability are stable, any solution that does not violate the constraints would be deemed viable. In the real world, however, both the available and required resources change unpredictably. Thus, the "robustness" of a solution to the sudden changes should be considered as a comparison criteria: this term can in turn represent the task scheduling problem as a minimization problem of a certain formula $L(n)$, which represents the load of all the tasks in a network n (i.e., global load).

A simplest form of $L(n)$ respecting all the described constraints can be $L(n) = \sum_{\forall x \in n} F(x)$, where $L(n)$ gives a unique numerical score to a target network n by adding up load scores of individual devices. To favor minimizing its value favors minimizing $L(n)$, $F(x)$ has to be a convex monotone function. For instance, let $F(x)$ be a quadratic function x^2 and n be composed of two devices loaded at 50% each. The global load would be $L(n) = 0.5^2 + 0.5^2 = 0.5$, while if just one of the devices is 100% loaded and the other one is unused the global load score would be higher: $L(n) = 1^2 + 0^2 = 1$. Having a uniform load distribution is preferable because, as stated earlier, minimizing this formula implies maximizing the average robustness of the devices to sudden load changes: since it is not possible to define a predictive model of the workload changes, we decided to assume a uniform distribution of workload change probability across all the devices. Under these assumptions, the optimal configuration has to be the one in which all the devices have the same (minimal) resource occupation, in terms of percentage value relative to maximum available resource of each device.

While it is not trivial to define the resource occupation that takes into account a number of incompatible measures (i.e., CPU, memory, network bandwidth usage), $F(x)$ should consider all these components at once for two main reasons. First, because depleting just one of these resources is enough to

[i] Although usually the data gets reduced from task i to task $i+1$, we encountered some cases in which the volume of data increases for different reasons, such as the necessity to add some metadata to speed up following processing steps, or because of the encapsulation needed to send the data down to the next module.

seriously degrade the performance of a device, thus every resource usage should be kept under control. Second, because considering all these aspects at once might help to efficiently use specialized devices; for example, if a network has a device with a huge amount of available memory it could be wise to delegate all the memory hungry tasks to that device.

Based on the prior researches analyzing this multi-criteria minimization problem, we decided to use the *simultaneous a priori optimization* presented in :[Hoo05] it simply consists in defining $F(x)$ as the weighted sum of all the components that have been decided to be considered. In our case this concept has slightly been modified in order to respect the convexity criteria described above, hence the linear weighed sum has been transformed into a weighted sum of the square of all the components, as follows: $F(x) = \alpha f_{cpu}(x)^2 + \beta f_{mem}(x)^2 + \gamma f_{net}(x)^2 + f_c(x)$. The first three components just represent formulas returning the resource usage ratio in terms of processing power, memory and network bandwidth: values are normalized, so that a value equal to 1 means that a certain resource is 100% used while 0 means that the resource is not being used at all. In addition, $f_c(x)$ is a component under control of the developer that can be used to enforce custom policies. The last $f_c(x)$ term may return either arbitrarily chosen constant values or a contiguous range of values and it is implementation dependent. For this reason its value is not squared nor weighted: the task of choosing appropriate values is left to the developer.

Here follows a more detailed description of the components:

- $f_{cpu}(x) = (\sum_{\forall i \in x} cpu_i \cdot bw_i)/cpu_{ref}$

  cpu_{ref} represents the bandwidth of the reference algorithm, while over the fraction sign lies the sum of the processing bandwidth of all the tasks being used in a device bw_i, weighted by the CPU usage ratio of each task cpu_i. This means that, for instance, a task with $cpu_i = 1$ (as fast as the reference algorithm) will need to have an incoming stream of input data with a bandwidth higher than cpu_{ref}, before overloading the CPU of the device. On the other hand, a task with $cpu_i = 2$ is two times slower than the reference task and, for this reason, the maximum incoming bandwidth it can have before causing CPU load problems is $cpu_{ref}/2$.

- $f_{mem}(x) = (\sum_{\forall i \in x} mem_i + bw_i \cdot mem_{\Delta i})/mem_{avail}$

  The $f_{mem}(x)$ component is easier to understand: the numerator of this fraction just calculates the total amount of memory currently used by all the active tasks, while mem_{avail} represents the total amount of available memory.

- $f_{net}(x) = max_n \left((\sum_{\forall i \in n} bw_i)/bw_n\right)$

  The $f_{net}(x)$ formula is slightly different from the previous ones, as it calculates a different score for each different communication channel n a device has and, then, it picks the maximum one.

- Finally, the $f_c(x)$ formula is an arbitrary value the algorithm adds to the score of a device in order to consider custom, additional policies. Formally, $f_c(x) = \sum_{\forall i \in active} p_i$, meaning that $f_c(x)$ is the sum of a series of penalties p_i belonging to the *active* set, where the *active* set is defined by the task scheduler depending on the policies that are being violated. Developers are able to add their own p_i coefficients, by writing the code that sets and clears them at will: for instance, this feature could be used to favor GPU-accelerated tasks to run in devices with compatible GPUs. By default, the framework uses this formula to mitigate the $f_{cpu}(x)$ score in case several processors are available.

9.2.4 The task scheduling algorithm

Now that we have a value that quantifies the cost of the distributed resource consumption, we have to present the algorithm that dynamically allocate tasks to devices, under the constraint to minimize

the overall cost of the distributed application.

The algorithm that minimizes the described load formula exploits the same working principles of MEDUSA:[BBSS] this algorithm is based on representing the load of each task as a cost, calculated by using the previously defined $F(x)$ formula as $cost_i = F(taskset) - F(taskset - i)$. Namely, $cost_i$ is the difference between load score of a device with all of its active tasks versus the score without the task i. This cost is then compared to a contract cost $contract_i$, representing how much extra load other devices are willing to accept.

For every task i currently scheduled to a device x, the load balancing algorithm continuously queries all its neighbors, in order to find the most convenient $contract_i$ value. If that value is lower than the $cost_i$ value the device x sustains, device x will try to delegate task i to the device providing that convenient $contract_i$. On the other side, devices receiving task delegation offers perform the same calculation and decide whether to accept the offer, to partially take the load, or to completely refuse the offer. When an offer is refused, the originating device tries with another device. This simple heuristic allows to develop a completely distributed algorithm on which each device takes care of moving its own tasks until no more "convenient exchanges" exists.

The $contract_i$ value was a fixed value in the MEDUSA model, while in our case it changes depending on each device and on each task, as it is calculated by using the usual $F(x)$ formula, just like $cost_i$. However, instead of considering the real set of tasks currently active in the target device, an arbitrary load coefficient is used in its place: for instance, if we assume that every other device is 80% loaded, the contract formula becomes $contract_i = F(t_{0.8} + i) - F(t_{0.8})$, where $t_{0.8}$ is a generic task taking exactly 80% of both available memory, CPU power and network bandwidth.

This choice has been made so that in this way a device can calculate, for a certain task, the contract costs of all the other devices by minimizing the amount of meta-data that have to be exchanged across the network to perform the calculation: every device just advertises its resource availabilities to everyone else at the beginning of the analysis, and then every other device can calculate its contract costs by assuming it is loaded by a certain fixed percentage.

It is important to note that this algorithm just implements an heuristic and that, by definition, cannot guarantee to find the optimal result. Depending on the order the task delegations are performed, in fact, the algorithm might reach the global optimum or it might remain stuck in a local optimum. However, as already proven in,[BBSS] as long as $F(x)$ is a monotonic growing convex function and the algorithm respects this "convenient trade" principle it is possible to have an high level of confidence about the good obtained values. Moreover, as stated previously, as long as the system finds solutions that are compatible to the various policies, it is not strictly required to have the best possible solution, but any "viable" solution (according to our metrics) can be considered enough to be able to use the framework without degrading the user experience of the owners of the different devices.

9.3 Architecture

This section presents the implementation of the methodology described in the previous section and the challenges we had to face in building the DELTA framework, which is written in Java. Figure 9.3 shows the architectural view of the DELTA framework, which depicts how a single DELTA instance running on a network device is internally structured.

Task scheduling. From the "logical application workflow", every device knows which tasks are currently being assigned to it and how these tasks are interconnected to each other. Based on the available resources of the device and the workload required by tasks, the *Task Scheduler* decides whether to accept a task or to delegate it to one of its neighbors. This decision is sent to the other devices encoded in XML messages, which allows all the devices of the network to known which tasks

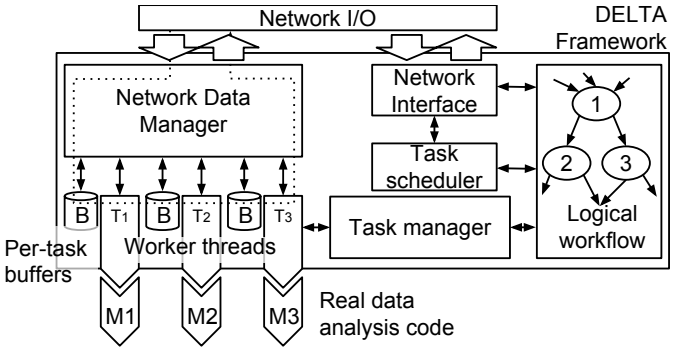

Figure 9.3: Architecture of the DELTA framework

are being assigned to which device. Currently this operation is carried out by means of a central communication server, but this can be changed in future implementations of the framework.

I/O management. Aside from the logical application workflow, there are the "real" tasks: a set of worker threads, one for each task, is created and handled by the framework. Each thread is connected to its own input buffer and to a piece of user-defined code, representing the real data analysis algorithm. A *Network Data Manager* module ensures that all the data received by the device is properly routed to the correct processing task and that the outputs of different tasks are forwarded to the proper target device(s).

Task management. The *Task Manager* constantly monitors the tasks running on the VM and calculates moving averages of the incoming traffic and the available resources. These estimates are then used for traffic smoothing when input data are bursty: if a device unexpectedly receives a huge block of data, the framework splits the block into smaller chunks and processes each of them at a time, at a rate that does not exceed the amount of the available resources. As long as enough input buffer space is provided, the device does not get overloaded and the subsequent tasks (and devices) will receive more stabilized volume of data. Even if the volume of input becomes consistently high and cannot be handled by the device (i.e., input queue gets backlogged) this scheme allows the device to buy time to delegate the task to a neighboring device with more available resources.

When implementing the DELTA framework we discovered a couple of problems that were not foreseen in the algorithm design step. Following sections present two extensions we designed (and implemented) in the DELTA framework in order to face those challenges.

9.3.1 Network delay resilient scheduling

Due to the well-known difficulties, it is impossible to have different components of a distributed system to react exactly the same time. Particularly, when tasks need to be migrated from one to another device, they cannot always be done instantaneously as several devices are involved in the job. If, for example, an input data stream is redirected from one device to another before the corresponding task is migrated (e.g, due to loss or delay of the task migration command), the input data should be lost until the task completes its migration. If the delay co-occurs at multiple locations in the workflow, the task schedulers may not decide which task to run locally and delegate to others, causing a network-wide oscillation.

While some network protocols (e.g., TCP) provide resiliency to traffic loss, unsynchronized task migration (due to delay needed to recover control packet loss) may still cause the cascaded task re-assignments. In order to solve this issue, instead of creating a complex, synchronized communication protocol, we simply impose a time delay δ before each task migration (i.e., allow "delayed" sync).

Under normal conditions, the task scheduler is supposed to update the logical workflow, according to the messages it receives from the network; the task manager, then, waits for δ to update the worker threads according to the logical workflow: this latency allows synchronization between real and logical workflow, preventing oscillations in the task scheduling. Additionally, the set of worker threads can also autonomously change: threads can be killed (or frozen) if they remain idle for too long to save resources, while new workers can be created if the device receives an input data stream targeted to a task that is not yet present in the logical application workflow. These modifications are reported to the logical application workflow by the Task Manager, again by keeping a certain latency between the modification events and the synchronization.

The additional latency introduced in the task migration process, and the fact that both the real and the logical workflow can change autonomously, represent the key components for creating an extremely stable platform: each device is able to properly work even without receiving task scheduling information from other devices at all; every data stream gets processed until it reaches the core even when task scheduling commands are undeliverable. Thanks to this feature the communication protocol used to take task scheduling decisions can be extremely simple, as it can safely ignore most of the potential synchronization issues among devices.

9.3.2 Transparent task migration

Another important consideration in task migration is how the *processing state* (e.g., the content of a TCP session table that is needed by a task to operate) is migrated across the network. First, not all tasks are stateful; for instance, this represents one of the information that are stored in the task manifest. To migrate a stateless task, our framework simply kills the corresponding threads in a device and launches a new copy of that task in the target device. To move a stateful tasks, our framework physically compresses their internal memory and physically moves it across devices to avoid interruptions on their processing. This can be done by exploiting the serialization primitives of Java objects, which allows each object to save its internal state, which is taken by DELTA and sent to a remote device for the proper de-serialization process. Additionally, as a robustness measure, the originating device keeps track of those stateful task migrations, so that it can forward any traffic arriving at the old device to the new instance of that task.

10

A case study: MOSAIC

Showing how the DELTA framework can be effective by using a powerful data analysis algorithm as use case

Contents

10.1 Integrating DELTA with MOSAIC

In order to evaluate the DELTA framework, we ported MOSAIC, an existing network analytic tool written in Java, to our system. MOSAIC is a set of data analysis algorithms that are able to associate the network traffic to the user device that generated it. For this purpose, it mines the information embedded in the network traffic to create, as the name suggests, "mosaics" that represent various categories of information about the given device. While MOSAIC supports mining of many different types of information, for the purpose of evaluating our framework, we configured it to look for geographic coordinates, social network IDs and the URLs browsed by each device.

MOSAIC overview.

In this configuration, the MOSAIC workflow can be split in three parts, as shown in Figure 10.1 (a). First, a probe (i.e., task (1) in Figure 10.1 (a)) that captures data packets from the traffic. Second, a sequence of independent stream processing tasks (i.e., tasks (2), (3), (4)), collectively termed "information extraction", which pre-processes the packets by aggregating them into data streams (flows) and extracting meaningful device information from them. Third, a final part termed "user correlation analysis" which attributes the data streams to identities of the originating devices (and users). This part represents the most complex portion of the algorithm because it needs to correlate different data streams generated by the same device from different networks (e.g., a device connected from a corporate network representing an office and an ISP representing home) over time.

Original MOSAIC.

In a typical setup, apart from the probe, all the processing parts reside in a data center. While this simplifies the implementation, the entire traffic probed from all the networks under analysis has to be replicated and delivered to the data center. Further, because the voluminous traffic needs to be handled, a large number of computation-intensive tasks is required to be distributed in the data center.

Noticing that the packets to data streams (task (2)), block generation (task (3)), and information extraction (task (4)) phases of the second part can operate on a subset of the network data, we focus on this part of MOSAIC for our jobs distribution algorithm. The porting of this part of MOSAIC to the DELTA framework allows to transform this tool in a distributed platform, where tasks from (1) to (4) are able to freely move across different processing devices available in the network under analysis. In a first instance, we kept each processing task "as is" in order to quantify the minimum amount of workload needed to distribute an existing algorithm such as MOSAIC. Then, in a second pass, as it will be presented in Section 10.2.6, we made an assumption that all the end user devices in the network are able to capture their own data, and we modified the algorithms by making them conscious about this assumption.

DELTA/MOSAIC.

As shown in Figure 10.1 (b), we had to implement a very limited set of changes to the MOSAIC workflow in order to port it to the DELTA framework. In fact, the only difference is that, now, several blocks are no longer assigned to a well-defined machine but they can be moved potentially everywhere. However, in practice some blocks have special requirements, hence some constraints exist for their placement. For instance, Tasks (2) and (3) are marked by the S flag, as they are *stateful*, while the C flag on Task (6) means that it can be executed only in the core. Finally, Task (7) is marked with the ST flag, meaning that just one instance of this module can exist, although it may not necessarily be ran in the core. These requirements are needed to be able to correctly perform the processing and to ensure that its output can be fed into the subsequent tasks of the MOSAIC algorithm, which are not shown in these figures for simplicity.

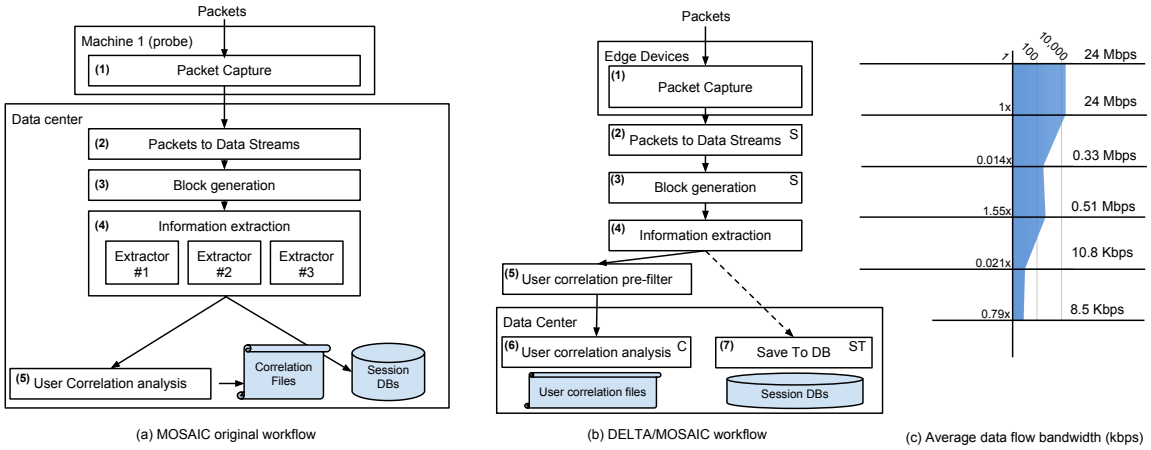

Figure 10.1: MOSAIC, distributed MOSAIC, and expected traffic reduction with the distributed MOSAIC

MOSAIC code porting.

The distributed version of MOSAIC also allows to assess the level of difficulties in porting the existing code to our framework. From the practical point of view, the porting required to create a set of "wrappers", one per each block, which were in charge of coordinating the send/receive operations between preceding and subsequent tasks. Since both DELTA and MOSAIC are written in the same programming language (i.e., Java), our wrappers were primarily in charge of calling the correct methods in the MOSAIC code, which was considered as a sort of external library. At the same time, the new wrappers were responsible to parse the input data, convert it to MOSAIC data objects, pass them to the original analytic and take the corresponding results, and finally send them to the next task with the help of Network Data Manager. This simple wrapping mechanism made the distribution of the MOSAIC to be extremely easy without requiring significant amount of extra coding: the wrapper is made with just a few hundred lines of code, which corresponds to about 0.4% of the length of the original MOSAIC code.

10.2 Performance results

10.2.1 Dataset description

In order to evaluate DELTA in a realistic workload, we use a network traffic trace collected from a medium-sized company network. The 24 hour-long trace was captured on a day in April 2013 and was properly anonymized. The network is composed by about 50 workstations connected via Ethernet, and by about the same amount of smartphones and tablets connected to a WiFi network. The captured file contains all the network data produced by these devices, including both local and the Internet traffic. On average, the capture file contains 470 Mbps data stream, which reduces to about 24 Mbps when considering only the traffic to/from the Internet. A tool has been used to read this capture file and to reproduce the traffic with the original timings. Additionally, the tool supports the possibility to customize the traffic generation speed, which was used to arbitrarily increase the traffic and simulate larger workloads while preserving the relative timings among packets.

The DELTA framework has been tested both by running multiple instances of its module in a single machine by running separate threads communicating through shared memory, and by using a

mixture of real and virtual machines with different capabilities. In the latter case, the components communicate by exchanging control packets on the Ethernet.

10.2.2 Evaluation of network overhead

We begin by demonstrating the possible reduction of network traffic sent to the data center when (part of) the network analytic is distributed across the network, possibly close to the edge of the target network. For this, we replay the network trace by simulating the activities of edge devices and measure the volume of the (reduced) traffic at different stages (tasks) shown in the distributed version of MOSAIC shown in Figure 10.1 (b).

Table 10.1: Amount of data sent to the remote analyzer

Traffic measurement point	Bandwidth
Initial traffic	24 Mbps
After packet filter	6 Mbps
After block (4) (theoretical)	8.3 Kbps
After block (4) (DELTA/MOSAIC)	10.8 Kbps

The results of measurements are summarized in Table 10.1. Based on our MOSAIC configuration, only DNS requests, HTTP and SSL/TLS data streams are actually passing the Packet Capture task (Task (1)). Given that the packet filter only captures these protocols and discard all others, the input stream in ingress to the subsequent MOSAIC tasks should be around 6 Mbps(second row of Table 10.1).

Further, it is safe to consider that not the entire information from the captured packets is needed for the subsequent analysis. Focusing on DNS names, online social network IDs, and geographic coordinates, MOSAIC algorithm is able to selectively extract only those information from the filtered traffic. Then the actual useful information is about 100 Bytes for every DNS request and 20 Bytes for social network IDs and geographic coordinates. Multiplied by the number of packets containing the information, the estimated amount of useful data gets reduced to about 8.3 Kbps (third row of Table 10.1).

In the real implementation, however, our measurements show that the data stream sent to the remote server is about 10.8 Kbps (fourth row of Table 10.1), which is pretty close to our estimates. The additional overhead is due to some implementation choices: for instance, messages are encoded in XML format that is rather verbose; also, additional information such as Device ID (from User Agent field of HTTP) are transported in the message to help facilitate User Correlation analysis.

Figure 10.1 (c) shows the reduction of data streams from task i to task $i + 1$: depending on the number of tasks executed locally, the network overhead drops to as low as 8.5 Kbps. In our distributed DELTA/MOSAIC setup, we executed tasks from (1) to (4) in edge devices, while task (5) was executed in the middle of the network (often in wired or wireless routers), and tasks (6) and (7) were executed in the data center. The reason for not executing task (5) in the edge device is due to its excessive consumption of computational resources, particularly CPU and main memory. Thus the DELTA algorithm finds more convenient to run this task in a less loaded, more capable device. In this configuration, we were able to consume about 10.8 Kbps of network bandwidth per user terminal, hence achieving an average data reduction of 99.96% compared to the initial input stream of 24 Mbps.

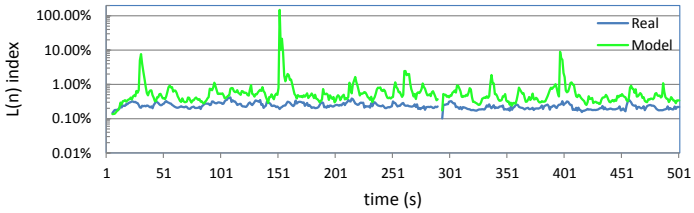

Figure 10.2: Global load of a network running DELTA/MOSAIC

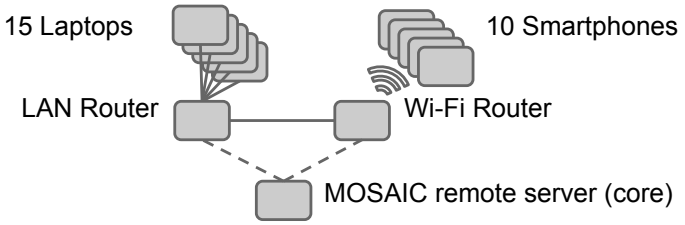

Figure 10.3: Deployment scenario of DELTA/MOSAIC

10.2.3 Efficiency of load distribution

Here we aim to provide the experimental evidences that the DELTA framework can distribute its tasks without negatively impacting performances of the network devices. This is important in that users would not accept a sensible worsening of the performance of their devices due to the additional overheads incurred by DELTA tasks running in the background.

To simulate large-scale networks (with many devices), we mainly simulated the corporate network environment in a dual-CPU server in which each network device (either a end-user host or a router) is made up of a set of threads. In addition to the setup, we ran a small-scale evaluation with real, physical hosts and virtual machines to ensure validity of the large-scale simulation.

The simulated network (shown in Figure 10.3) includes 25 edge devices, 2 intermediate routers and a single core unit representing the data center running MOSAIC User Correlation analysis. Among the 25 edge devices, 10 of them are considered to be smartphones with very limited resources, while the remaining 15 are considered to be workstations. Each device replays the data trace at a rate five times faster than the original thus simulating 120 Mbps per device.

Figure 10.2 shows the behavior of the global load $\mathbf{L}(n)$ presented in Section 9.2, which is a number that takes into account the additional CPU, memory and network consumption due to the DELTA jobs on the different network devices where those jobs are running[i]. Figure 10.2 shows two plots, namely "model" and "real", which refer to the $\mathbf{L}(n)$ computed by the DELTA model and the $\mathbf{L}(n)$ actually measured from the network, respectively.

While the theoretical $\mathbf{L}(n)$ exhibits a few spikes over time, the global load in the real system only shows little variation. The (more desirable) reduction in the variation is achieved by the Task Manager (c.f., Section 9) which smoothed the traffic out: when bursty data arrive, the system subdivides and buffers them.

[i]It is worthy mentioning that an $\mathbf{L}(n)$ index equal to 3 (translated to 100% in the figure) means that all the available CPU, memory and network resources have been consumed by the distributed algorithm; the lowest the number, the smaller the impact of the distributed jobs across the network devices.

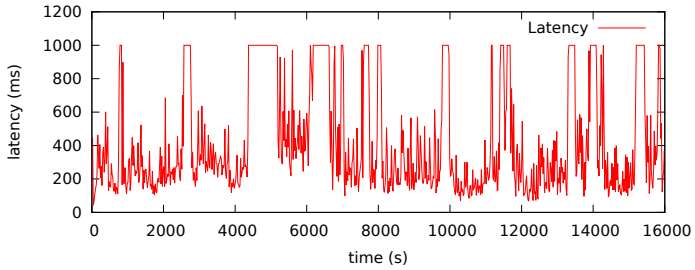

Figure 10.4: Processing latency of DELTA/MOSAIC in a 1Ghz 1-core virtual machine

10.2.4 Effectiveness of the task scheduler

As the task scheduling in DELTA is provably NP-hard, Task Scheduler in each device uses a greedy heuristic for task assignment. While this has clear advantages in terms of scalability, it may lead to locally optimal decisions because in real world, the assignment of the tasks is dependent on the order of their arrival.

This section aims at evaluating the optimality of the current DELTA task scheduler. For this, we developed a reference algorithm that runs Simulated Annealing[KGV83] on each second of the trace and compares its $min\{\mathbf{L}(n)\}$ to that of current DELTA heuristic. The probabilistic search algorithm shows that, in the best case, the $\mathbf{L}(n)$ index obtained by DELTA task scheduler was only 1% higher that the best alternative solution found from the simulated annealing. Based on the findings, in spite of its greedy nature, we believe our task scheduling algorithm is able to quickly converge to good overall solutions. As a consequence, the additional overhead introduced by a more sophisticated task scheduler algorithm may not be justified at least in our use case.

10.2.5 Measurement of processing latency

In order to analyze how DELTA tunes the CPU load of a device, a virtual machine has been created to resemble a low-end device (a single core 1Ghz processor, 2GB RAM). In this experiment, the device has been required to analyze our entire 24-hours capture at 5x speed. Every processing task is executed every second and it should try to empty its input buffer. If the task requires less than one second to complete, it remains idle for the remaining time, thus freeing up the CPU to perform other operations.

In order to precisely measure the CPU occupancy of DELTA/MOSAIC, every processing task running on the device under measurement has been synchronized, so that all the tasks start processing their data at the same time, and then at each iteration the time required by all the tasks to complete has been measured.

Figure 10.4 shows the latency measurement results. As can be seen from the maximum latency upper bounded by 1000 ms, the CPU usage of DELTA is strictly limited. Even in this very restrictive situation, the CPU remains idle for the most of the time (for 79% of the time). There are two main reasons for this excellent behavior: first, as the network traffic is processed in semi-real time, the amount of data the tool has to process every second is relatively small, apart from few instants where the traffic suddenly surges for short periods of time; second, thanks to the CPU limiter, even during these spikes the system does not get overloaded, it simply fills up the buffers. Moreover, it must be considered that in this experiment a single device had to analyze the entire traffic trace. In a real

environment, thanks to the distributed flavor of DELTA/MOSAIC, this device would have had to just process its own traffic, thus making CPU idle times even higher.

10.2.6 Improvements by accessing to richer data

As our capturing probes are distributed and become very close to the end systems (i.e., the network traffic producers), the last experiment focuses on measuring the benefits of this privileged position of the analyzers. Particularly, we considered the block (3) of Figure 10.1b, which associates each traffic information with the user device that generated the data.

In this test we assumed the best-case scenario, i.e., the case in which each user terminal connected to the wired network is also capable of capturing and pre-processing data, as well as the Wi-Fi routers[ii]. In this way, cabled user terminals are able to capture their own traffic and associate to each data stream an unique session ID, while Wi-Fi routers can distinguish the different devices based on their MAC addresses.

The MOSAIC algorithm has been optimized to exploit these modifications to identify devices by replacing the original algorithm in block (3) with a new, trivial "block generation" mechanism.

The first result is, as expected, the capability of this block to associate the 100% to the proper user terminal (for instance, with 100% accuracy) compared to the original 69% of the users with 96.4% of accuracy shown in.[XSL+13] The second result refers to the usage consumption of this block. In fact, the original algorithm was extremely expensive in terms of required processing resources, as (alone) it responsible for more than the 65% of the total CPU time required to analyze our sample data. Instead, the new algorithm allowed to reduce the CPU consumption of block (3) up to 85%, hence greatly reducing the workload of the MOSAIC application and further increasing the possibility to distribute the analytics on devices with reduced processing capabilities.

10.2.7 Summary

Our experiments shown that, with minimal modifications to the code, it is possible to use the DELTA framework to distribute a portion of the MOSAIC data analysis framework across the physical devices that reside in network under analysis.

Advantages are noticeable: the amount of data that has to be sent to the datacenter for further analysis were reduced to the 0.2% of the amount of traffic generated across the network, while at the same time the precision of the analytics was improved thanks to the fact that our probes have access to "better" data, with more valuable information. For instance, the capability to associate each information extracted from the traffic to the proper user terminal rose from 69% to 100%, meaning that all the extracted information is now associated to the correct generating device. Furthermore, the framework has been proved being very reactive to sudden workload changes, with the task scheduling able to relocate all the jobs in very short amount of time. This can be seen in Figure 10.2: even if at second 60 and second 150 we have artificially introduced two huge workload changes, by increasing the data production speed, the load spikes are only visible in the theoretical load line while being barely noticeable in the real load. This is because, at first, the workload spike is absorbed by buffers while, after few instants, the task scheduling algorithms redirect the flows to devices able to sustain that load. Finally, since running additional software on user terminal may rise objections in particular on mobile

[ii] We assume that WiFi devices are mostly mobile terminals, hence our algorithm tend to avoid to load some additional software on them in order to preserve their battery; as a consequence, the algorithm tend to activate that software on the nearest router.

devices where energy consumption is a big issue, we would like to mention that our CPU limiter further improves the adaptability of the system, by keeping the resource usage down to reasonable amounts even when a device has to face a sudden workload change.

Part III

Conclusion

11

Conclusion

.

Contents

At the beginning of this thesis a series of methodologies was proposed to take care of the problematics that were limiting the adopotion of both formal methods and traffic monitoring/analysis solutions in work environments. For what concerns fromal methods, the proposed methodology aimed to be easily usable by non-specialized developers, by allowing to use widely known programming languages to model cryptographic protocols and by automatically taking care of the most critical aspects of the model-driven development work flow; for what concerns data analysis, instead, the proposed methodologies aimed at reducing deployment costs either by improving performance of data processing on general-purpose devices and by proposing tools to automatically redistribute workloads among existing devices to efficiently reuse available resources and achieve better results.

From a practical point of view, the proposed methodologies concretized in the development of three tools: the JavaSPI framework, for the model-driven development of security protocols through Java models, an improved version of the iNFAnt tool for regexp-based packet processing, accompanied by a series of additional toolsets needed to perform multi-stride opimizations and, finally, the DELTA Framework, a library useful to develop automatically distributed traffic analysis algorithms. Efficacy of the tools have been demonstrated by means of use cases: real world problems have been solved by using these tools, by measuring their impact in terms of development effort, performance and deployment footprint. The next sections resume the main results obtained by each tool, while the last one will present some of our ideas about how these tools could further improve in the future.

11.1 JavaSPI

The JavaSPI Framework managed to reach all the objectives posed for what concerns simplifying the development of safe cryptographic security protocols through formal methods: by using one of the possible SSL 3.0 handshake configurations, a protocol widely known with a lot of commercial implementations, it has been shown how, thanks to JavaSPI, it has been possible to design the protocol model, build formal proofs about its security properties and convert it to an implementation able to interoperate with other commercially available solutions. The model is slightly bigger with respect to the ones built by directly using formal languages due to the verbosity of the Java language, but it still consists on few hundreds of lines of code, way less than the thousands of lines composing commercial implementations, and moreover it is much easier to understand, thanks to the fact that it describes the protocol at high level, without delving in all the implementation details. Thanks to the inheritance annotation system, adding implementation details just required to annotate 10% of the code, thus keeping it very readable.

From a performance standpoint, it has been proven that in the environment of cryptographic security protocols the main claim against code generation, regarding the fact that since the generation is automated it is not possible to properly optimize the code for performance, is not that relevant: the computational bottleneck of these algorithms, in fact, just regards the cryptography operations, thus the performance of a cryptographic protocol implementation are typically just determined by the speed of the used cryptographic library, while algorithm inefficiencies might easily become negligible. In the case of SSL, for instance, we compared performance of our implementation with respect to JSSE, a tool using exactly the same Java crytpographic library used by our implementation. Even if the JSSE code is surely more optimized than our automatically generated one, the performance difference does not exceed 5%.

11.2 Traffic analysis

Concerning traffic analysis, at first we focused on a single data analysis tool, iNFAnt, by improving its performance, then we proposed DELTA as a tool to improve, more in general, performance of traffic analysis algorithms not just in terms of processing bandwidth but, more importantly, also in terms of information extraction power by reducing, at the same time, the amount of data that have to be sent to the central analyzer.

The new multi-stide and multi-map algorithms proposed in this thesis managed to improve iNFAnt processing performance by making such promising techniques applicable to real world environments, where size of regular expression rule sets was too big to make the previously known multi-stride algorithms working properly. Numerically speaking, the throughput boost obtained with these new algorithms, in respect to the results obtainable with previous state-of-the-art algorithms, ranges about 4x for what concerns medium-sized NFAs and 3x with big sized NFAs.

Analyzing the traffic produced by a medium-sized company (about 50 employees) during a 24 hours time span by using a traffic analysis tool called MOSAIC has been used as use case to demonstrate the benefits of using the DELTA framework. The original deployment solution consisted in using probes to send all the traffic produced by the network to a remote data center performing the analysis: this operation required a constant 24Mbps data transfer flow, while the algorithm was able to identify 69% of users with an accuracy of 96.4%. Thanks to the DELTA framework, few hundreds of additional lines of code have been enough to build a distributed version of MOSAIC: in the hypothesis that all the end devices and the intermediate routers in the corporate network were involved in the traffic analysis, DELTA/MOSAIC required to send to the remote data centere a data flow of just about 10Kbps (99.95% reduction) while, at the same time, it allowed to obtain a 100% user recognition rate with 100% accuracy. Finally, the additional overhead posed to the devices was close to be neglibible: even the slowest devices considered (devices with an old single core 1Ghz processor and just 2GB of RAM) managed to take part of the processing while leaving the processor in idle state during 79% of the time, and this overhead was automatically reduced by the framework when the devices were used to perform other tasks.

11.3 Conclusions and future works

In conclusion, the tools and techniques proposed throughout this thesis have proven to be very promising: applying these approaches to real use cases is expected to undoubtedly provide several benefits, while at the same time there is still plenty of space to furhter improve the capabilities of these tools and approaches: this thesis proposes new methodologies and ideas, but even if the benefits of these approaches are already evident, As example, the compression efficiency of the multi-map multi-stride approach is currently limited by a very simple heuristic that, in spite of the fact that it is already able to provide very good results, could be further refined in order to provide even better results. A similar reasoning can be performed concerning the JavaSPI and DELTA frameworks: both these tools could be further refined by extending their capabilities and by making them more powerful and easy to use; in the particular case of the DELTA framework a further refinement is expected to be performed in order to merge its algorithms with the programmable routers presented in.[CPR13]

The general idea is that this thesis aims to solve some problems by proposing new approaches that, by design, provides better results with respect to traditional techniques. However, this is just a starting

point, as there is plenty of space to do even better. I can't wait to see how things will evolve during time...

Bibliography

[AN96] Martín Abadi and Roger Needham. Prudent engineering practice for cryptographic protocols. *IEEE Transactions on Software Engineering*, 22(1):6–15, 1996.

[APS14] Matteo Avalle, Alfredo Pironti, and Riccardo Sisto. Formal verification of security protocol implementations: a survey. *Formal Aspects of Computing*, 6(1):99–123, 2014.

[APSP11] Matteo Avalle, Alfredo Pironti, Riccardo Sisto, and Davide Pozza. The java spi framework for security protocol implementation. In *Availability, Reliability and Security (ARES), 2011 Sixth International Conference on*, pages 746–751. IEEE, 2011.

[ARS12] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. Efficient multistriding of large non-deterministic finite state automata for deep packet inspection. In *Communications (ICC), 2012 IEEE International Conference on*, pages 1079–1084. IEEE, 2012.

[ARSBar] Matteo Avalle, Fulvio Risso, Han Song, and Mario Baldi. Pushing network analytics toward the edge of the network. to appear.

[ARSar] Matteo Avalle, Fulvio Risso, and Riccardo Sisto. Scalable algorithms for nfa multi-striding and nfa-based deep packet inspection on gpus. to appear.

[BBSS] Magdalena Balazinska, Hari Balakrishnan, Jon Salz, and Mike Stonebraker. The Medusa Distributed Stream-Processing System.

[BC07] Michela Becchi and Patrick Crowley. An improved algorithm to accelerate regular expression evaluation. In *Proceedings of the 3rd ACM/IEEE Symposium on Architecture for networking and communications systems*, pages 145–154. ACM, 2007.

[BC08] Michela Becchi and Patrick Crowley. Efficient regular expression evaluation: theory to practice. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 50–59. ACM, 2008.

[BC13] Michela Becchi and Patrick Crowley. A-dfa: A time- and space-efficient dfa compression algorithm for fast regular expression evaluation. *ACM Trans. Archit. Code Optim.*, 10(1):4:1–4:26, 2013.

[BFC08] Michela Becchi, Mark Franklin, and Patrick Crowley. A workload for evaluating deep packet inspection architectures. In *Proceedings of the 2008 IEEE International Symposium on Workload Characterization.* IEEE, 2008.

[BFGT08] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. *ACM Transactions on Programming Languages and Systems*, 31(1):1–61, 2008.

[Bla09a] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 17(4):363–434, 2009.

[Bla09b] Bruno Blanchet. Automatic verification of correspondences for security protocols. *Journal of Computer Security*, 2009.

[BPP03] G M Bierman, M J Parkinson, and A M Pitts. Mj: An imperative core calculus for java and java with effects. Technical Report 563, 2003.

[BTC06] B.C. Brodie, D.E. Taylor, and R.K. Cytron. A scalable architecture for high-throughput regular-expression pattern matching. In *ACM SIGARCH Computer Architecture News*, volume 34, pages 191–202. IEEE, 2006.

[CPR13] I. Cerrato, M. Pramotton, and F. Risso. Moving applications from the host to the network: Experiences, challenges and findings. In *Communications Workshops (ICC), 2013 IEEE International Conference on*, pages 744–749, 2013.

[CRRS10] Niccolo' Cascarano, Pierluigi Rolando, Fulvio Risso, and Riccardo Sisto. infant: Nfa pattern matching on gpgpu devices. *SIGCOMM Comput. Commun. Rev.*, 40(5):20–26, 2010.

[DR08] Tim Dierks and Eric Rescorla. The transport layer security (tls) protocol version 1.2. RFC 5246, 2008.

[DY83a] D. Dolev and a. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[DY83b] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[GM84] Shafi Goldwasser and Silvio Micali. Probabilistic encryption. *Journal of Computer and System Sciences*, 28(2):270 – 299, 1984.

[GSD+09] F Gringoli, Luca Salgarelli, M Dusi, N Cascarano, F Risso, and k. c. Claffy. GT: picking up the truth from the ground for internet traffic. *SIGCOMM Comput. Commun. Rev.*, 39(5):12–18, October 2009.

[Hoo05] H Hoogeveen. Multicriteria scheduling. *European Journal of operational research*, 2005.

[KGV83] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[KSE08] Shijin Kong, Randy Smith, and Cristian Estan. Efficient signature matching with multiple alphabet compression tables. *Proceedings of the 4th international conference on Security and privacy in communication netowrks - SecureComm '08*, page 1, 2008.

[Low95] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56(3):131 – 133, 1995.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[PJ09] Alfredo Pironti and J Jürjens. Black-box monitoring of security protocols, revision. Technical report, 2009.

[PS07] Alfredo Pironti and Riccardo Sisto. An experiment in interoperable cryptographic protocol implementation using automatic code generation. In *Computers and Communications, 2007. ISCC 2007. 12th IEEE Symposium on*, pages 839–844. IEEE, 2007.

[PS10] Alfredo Pironti and Riccardo Sisto. Provably correct java implementations of spi calculus security protocols specifications. *Computers & Security*, 29(3):302–314, 2010.

[PS12] Alfredo Pironti and Riccardo Sisto. Safe abstractions of data encodings in formal security protocol models. *Formal Aspects of Computing*, pages 1–43, 2012.

[PSD04] Davide Pozza, Riccardo Sisto, and Luca Durante. Spi2java: Automatic cryptographic protocol Java code generation from spi calculus. In *18th International Conference on Advanced Information Networking and Applications (AINA)*, pages 400–405, 2004.

[THW02] H Topcuoglu, S Hariri, and M Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems Journal*, 2002.

[XSL+13] Ning Xia, Han Hee Song, Yong Liao, Marios Iliofotou, Antonio Nucci, Zhi-Li Zhang, and Aleksandar Kuzmanovic. Mosaic: quantifying privacy leakage in mobile networks. In *Proceedings of the ACM SIGCOMM 2013*, pages 279–290, New York, NY, USA, 2013. ACM.

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *23rd Annual Symposium on Foundations of Computer Science (FOCS)*, pages 80–91, 1982.

[YKGS11] Liu Yang, Rezwana Karim, Vinod Ganapathy, and Randy Smith. Fast, memory-efficient regular expression matching with nfa-obdds. *Computer Networks*, 55(15):3376–3393, 2011.

A

Appendix

Complete JavaSPI evolution rules

A.1 JavaSPI evolution rules

JavaSPI τ evolution rules regarding packet composition / decomposition and other accessory rules:

$$
\begin{array}{rcl}
`final\ Type\ t = new\ Type(data);'\ P, \sigma & \xrightarrow{\tau^*}_J & P, \\
 & & \{t \to M\} \cup \sigma \\
 & & \\
`final\ Pair < A, B > p = new\ Pair(a, b);'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{a \to M\} \cup \{b \to N\} \in \sigma & & \{p \to (M, N)\} \cup \sigma \\
 & & \\
`final\ Integer\ s = new\ Integer(x);'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{x \to M\} \in \sigma & & \{s \to suc(M)\} \cup \sigma \\
 & & \\
`final\ Type\ a = p.getLeft();'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{p \to (M, N)\} \in \sigma & & \{a \to M\} \cup \sigma \\
 & & \\
`final\ Type\ b = p.getRight();'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{p \to (M, N)\} \in \sigma & & \{b \to N\} \cup \sigma \\
 & & \\
`fail();', \sigma & \xrightarrow{\tau^*}_J & 0, \sigma \\
`event(\text{"}name\text{"}, params);'\ P, \sigma & \xrightarrow{\tau^*}_J & P, \sigma
\end{array}
$$

JavaSPI τ evolution rules regarding cryptographic operations

$$
\begin{array}{rcl}
`final\ Hashing\ h = new\ Hashing(a);'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{a \to M\} \in \sigma & & \{h \to H(M)\} \cup \sigma \\
 & & \\
`final\ SharedKey\ sk = new\ SharedKey(a);'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{a \to M\} \in \sigma & & \{sk \to M \sim\} \cup \sigma \\
 & & \\
`final\ KeyPair\ kp = new\ KeyPair(a);'\ P, & \xrightarrow{\tau^*}_J & P, \\
\{a \to M\} \in \sigma & & \{kp \to (M+, M-)\} \cup \sigma \\
 & & \\
`final\ SharedKeyCiphered < Type > x & \xrightarrow{\tau^*}_J & P, \\
= new\ SharedKeyCiphered(k, m)'\ P, & & \{x \to \{N\}_{M\sim}\} \cup \sigma \\
\{k \to M \sim\} \cup \{m \to N\} \in \sigma & & \\
 & & \\
`final\ PrivateKeyCiphered < Type > x & \xrightarrow{\tau^*}_J & P,
\end{array}
$$

$$
\begin{array}{rcl}
= new\ PrivateKeyCiphered(k,m)'\ P, & & \{x \to [\{N\}]_{M-}\} \cup \sigma \\
\{k \to M-\} \cup \{m \to N\} \in \sigma & & \\
\\
`final\ PublicKeyCiphered < Type > x & \overset{\tau *}{\to}_J & P, \\
= new\ PublicKeyCiphered(k,m)'\ P, & & \{x \to [\{N\}]_{M+}\} \cup \sigma \\
\{k \to M+\} \cup \{m \to N\} \in \sigma & & \\
\\
`final\ Type\ t = x.decrypt();'\ P, & \overset{\tau *}{\to}_J & P, \\
\{k \to M \sim\} \cup \{x \to \{N\}_{M\sim}\} \in \sigma & & \{t \to N\} \cup \sigma \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{M-}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{M+}\} \in \sigma & & \\
\\
`final\ Type\ t = x.decrypt();'\ P, & \overset{\tau *}{\to}_J & `fail();', \sigma \\
\{k \to M \sim\} \cup \{x \to \{N\}_{O\sim}\} \in \sigma & & \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{O-}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{O+}\} \in \sigma & & \\
\\
`final\ RC < Type >\ rc = x.decrypt\_w(k);'\ P, & \overset{\tau *}{\to}_J & P, \\
\{k \to M \sim\} \cup \{x \to \{N\}_{M\sim}\} \in \sigma & & \{rc \to (TRUE, N)\} \cup \sigma \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{M-}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{M+}\} \in \sigma & & \\
\\
`final\ RC < Type >\ rc = x.decrypt\_w(k);'\ P, & \overset{\tau *}{\to}_J & P, \\
\{k \to M \sim\} \cup \{x \to \{N\}_{O\sim}\} \in \sigma & & \{rc \to (FALSE, NULL)\} \cup \sigma \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{O-}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{O+}\} \in \sigma & & \\
\\
`final\ Type\ t = rc.getValue();'\ P, & \overset{\tau *}{\to}_J & P, \\
\{rc \to (TRUE, N)\} \in \sigma & & \{t \to N\} \cup \sigma
\end{array}
$$

JavaSPI τ evolution rules regarding conditional statements

$$
\begin{array}{rcl}
`if(cond)\{'\ P\ `\}', & \overset{\tau *}{\to}_J & P, \sigma \\
cond = `a.equals(b)' \wedge \{a \to M\} \cup \{b \to M\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (TRUE, M)\} \in \sigma & & \\
\\
`if(cond)\{'\ P\ `\}', & \overset{\tau *}{\to}_J & `fail();', \sigma \\
cond = `a.equals(b)' \wedge \{a \to M\} \cup \{b \to N\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (FALSE, NULL)\} \in \sigma & & \\
\\
`if(cond)\{'\ P\ `\}else\{'\ Q\ `\}', & \overset{\tau *}{\to}_J & P, \sigma \\
cond = `a.equals(b)' \wedge \{a \to M\} \cup \{b \to M\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (TRUE, M)\} \in \sigma & & \\
\\
`if(cond)\{'\ P\ `\}else\{'\ Q\ `\}', & \overset{\tau *}{\to}_J & Q, \sigma \\
cond = `a.equals(b)' \wedge \{a \to M\} \cup \{b \to N\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (FALSE, NULL)\} \in \sigma & &
\end{array}
$$

JavaSPI λ evolution rules

$$
\begin{array}{rcl}
`c.send(o);'\ P, & \overset{c!M}{\to}_J & P, \sigma \\
\{o \to M\} \in \sigma & &
\end{array}
$$

$$\frac{\text{`}final\ Type\ t = c.receive(Type.class);'\ P,\ \sigma \quad \overset{c?M}{\rightarrow}_J \quad P,\ \{t \rightarrow M\} \cup \sigma}{}$$

A.2 Concrete Java evolution rules

Java τ evolution rules regarding packet composition / decomposition and other accessory rules:

$$\begin{array}{rcl}
\text{`}final\ TypeCC\ t = new\ TypeCC(data, par);'\ P, \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{t \rightarrow M^{par}\} \cup \sigma \\
\text{`}final\ Pair < A, B > p = new\ Pair(a, b);'\ P,\ \{a \rightarrow M^{par}\} \cup \{b \rightarrow N^{par'}\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{p \rightarrow (M^{par}, N^{par'})\} \cup \sigma \\
\text{`}final\ Integer\ s = new\ Integer(x);'\ P,\ \{x \rightarrow M^{par}\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{s \rightarrow suc(M^{par})\} \cup \sigma \\
\text{`}final\ Type\ a = p.getLeft();'\ P,\ \{p \rightarrow (M^{par}, N^{par'})\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{a \rightarrow M^{par}\} \cup \sigma \\
\text{`}final\ Type\ b = p.getRight();'\ P,\ \{p \rightarrow (M^{par}, N^{par'})\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{b \rightarrow N^{par'}\} \cup \sigma \\
\text{`}fail();', \sigma & \overset{\tau^*}{\rightarrow}_J & 0, \sigma \\
\text{`}event(\text{"}name\text{"}, params);'\ P, \sigma & \overset{\tau^*}{\rightarrow}_J & P, \sigma
\end{array}$$

Java τ evolution rules regarding cryptographic operations

$$\begin{array}{rcl}
\text{`}final\ HashingCC\ h = new\ HashingCC(a, par);'\ P,\ \{a \rightarrow M\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{h \rightarrow H(M)^{par}\} \cup \sigma \\
\text{`}final\ SharedKey\ sk = new\ SharedKey(a, par);'\ P,\ \{a \rightarrow M\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{sk \rightarrow M \sim^{par}\} \cup \sigma \\
\text{`}final\ KeyPair\ kp = new\ KeyPair(a, par);'\ P,\ \{a \rightarrow M\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{kp \rightarrow (M+^{par}, M-^{par})\} \cup \sigma \\
\text{`}final\ SharedKeyCiphered < Type > x = new\ SharedKeyCiphered(k, m, par)'\ P,\ \{k \rightarrow M \sim\} \cup \{m \rightarrow N\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{x \rightarrow \{N\}^{par}_{M\sim}\} \cup \sigma \\
\text{`}final\ PrivateKeyCiphered < Type > x = new\ PrivateKeyCiphered(k, m, par)'\ P,\ \{k \rightarrow M-\} \cup \{m \rightarrow N\} \in \sigma & \overset{\tau^*}{\rightarrow}_J & P,\ \{x \rightarrow [\{N\}]^{par}_{M-}\} \cup \sigma \\
\text{`}final\ PublicKeyCiphered < Type > x = new\ PublicKeyCiphered(k, m, par)'\ P, & \overset{\tau^*}{\rightarrow}_J & P,\ \{x \rightarrow [\{N\}]^{par}_{M+}\} \cup \sigma
\end{array}$$

$$\{k \to M+\} \cup \{m \to N\} \in \sigma$$

$$\begin{array}{rll}
`final\ Type\ t = x.decrypt(k, par);' \ P, & \xrightarrow{\tau *}_J & P, \\
\{k \to M \sim^{par'}\} \cup \{x \to \{N\}^{par}_{M\sim^{par'}}\} \in \sigma & & \{t \to N\} \cup \sigma \\
or \{k \to M-^{par'}\} \cup \{x \to [\{N\}]_{M-^{par'}}\}^{par} \in \sigma & & \\
or \{k \to M+^{par'}\} \cup \{x \to \{[N]\}_{M+^{par'}}\}^{par} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`final\ Type\ t = x.decrypt(k, par);' \ P, & \xrightarrow{\tau *}_J & `fail();', \sigma \\
\{k \to M \sim\} \cup \{x \to \{N\}_{O\sim}\} \in \sigma & & \\
or \{k \to M \sim^{par'}\} \cup \{x \to \{N\}^{par''}_{M\sim^{par'}}\} \in \sigma & & \\
or \{k \to M \sim^{par'}\} \cup \{x \to \{N\}^{par}_{M\sim^{par''}}\} \in \sigma & & \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{O-}\} \in \sigma & & \\
or \{k \to M-^{par'}\} \cup \{x \to [\{N\}]^{par''}_{M-^{par'}}\} \in \sigma & & \\
or \{k \to M-^{par'}\} \cup \{x \to [\{N\}]^{par}_{M-^{par''}}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{O+}\} \in \sigma & & \\
or \{k \to M+^{par'}\} \cup \{x \to \{[N]\}^{par''}_{M+^{par'}}\} \in \sigma & & \\
or \{k \to M+^{par'}\} \cup \{x \to \{[N]\}^{par}_{M+^{par''}}\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`final\ RC < Type >\ rc = x.decrypt_w(k);' \ P, & \xrightarrow{\tau *}_J & P, \\
\{k \to M \sim\} \cup \{x \to \{N\}_{M\sim}\} \in \sigma & & \{rc \to (TRUE, N)\} \cup \sigma \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{M-}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{M+}\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`final\ RC < Type >\ rc = x.decrypt_w(k);' \ P, & \xrightarrow{\tau *}_J & P, \\
\{k \to M \sim\} \cup \{x \to \{N\}_{O\sim}\} \in \sigma & & \{rc \to (FALSE, NULL)\} \\
or \{k \to M \sim^{par'}\} \cup \{x \to \{N\}^{par''}_{M\sim^{par'}}\} \in \sigma & & \cup\sigma \\
or \{k \to M \sim^{par'}\} \cup \{x \to \{N\}^{par}_{M\sim^{par''}}\} \in \sigma & & \\
or \{k \to M-\} \cup \{x \to [\{N\}]_{O-}\} \in \sigma & & \\
or \{k \to M-^{par'}\} \cup \{x \to [\{N\}]^{par''}_{M-^{par'}}\} \in \sigma & & \\
or \{k \to M-^{par'}\} \cup \{x \to [\{N\}]^{par}_{M-^{par''}}\} \in \sigma & & \\
or \{k \to M+\} \cup \{x \to \{[N]\}_{O+}\} \in \sigma & & \\
or \{k \to M+^{par'}\} \cup \{x \to \{[N]\}^{par''}_{M+^{par'}}\} \in \sigma & & \\
or \{k \to M+^{par'}\} \cup \{x \to \{[N]\}^{par}_{M+^{par''}}\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`final\ Type\ t = rc.getValue();' \ P, & \xrightarrow{\tau *}_J & P, \\
\{rc \to (TRUE, N)\} \in \sigma & & \{t \to N\} \cup \sigma
\end{array}$$

Java τ evolution rules regarding conditional statements

$$\begin{array}{rll}
`if(cond)\{'\ P\ `\}', & \xrightarrow{\tau *}_J & P, \sigma \\
cond = `a.equals(b)' \wedge \{a \to M^{par}\} \cup \{b \to M^{par}\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (TRUE, M)\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`if(cond)\{'\ P\ `\}', & \xrightarrow{\tau *}_J & `fail();', \sigma \\
cond = `a.equals(b)' \wedge \{a \to M\} \cup \{b \to N\} \in \sigma & & \\
orcond = `a.equals(b)' \wedge \{a \to M^{par}\} \cup \{b \to M^{par'}\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (FALSE, NULL)\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll}
`if(cond)\{'\ P\ `\}else\{'\ Q\ `\}', & \xrightarrow{\tau *}_J & P, \sigma \\
cond = `a.equals(b)' \wedge \{a \to M^{par}\} \cup \{b \to M^{par}\} \in \sigma & & \\
or\ cond = `rc.isValid()' \wedge \{rc \to (TRUE, M)\} \in \sigma & &
\end{array}$$

$$\begin{array}{rll} \text{`}if(cond)\{' \ P \ \text{`}\}else\{' \ Q \ \text{`}\}', & \overset{\tau *}{\rightarrow}_J & Q, \sigma \\ cond = \text{`}a.equals(b)' \wedge \{a \rightarrow M\} \cup \{b \rightarrow N\} \in \sigma & & \\ or\ cond = \text{`}a.equals(b)' \wedge \{a \rightarrow M^{par}\} \cup \{b \rightarrow M^{par'}\} \in \sigma & & \\ or\ cond = \text{`}rc.isValid()' \wedge \{rc \rightarrow (FALSE, NULL)\} \in \sigma & & \end{array}$$

Java λ evolution rules

$$\begin{array}{rll} \text{`}c.send(o);' \ P, & \overset{c!M}{\rightarrow}_J & P, \sigma \\ \{o \rightarrow M^{par}\} \in \sigma & & \\ & & \\ \text{`}final\ Type\ t = c.receive(Type.class);' \cup P, & \overset{c?M}{\rightarrow}_J & P, \\ \sigma & & \{t \rightarrow M^{par}\} \cup \sigma \end{array}$$

A.3 $J()$ translation rules

Under the assumption that $J(\sigma) = \sigma$ translation rules are defined only on P, while σ have been omitted.

Translation of statements regarding packet composition / decomposition and other accessory rules:

$$\begin{array}{rcl} J(\text{`}final\ Type\ t = new\ Type(data);' \ P) & \rightarrow & \text{`}final\ TypeCC\ t = \\ & & new\ TypeCC(data, params);' \ J(P) \\ & & \\ J(\text{`}final\ Pair < A, B > p = & \rightarrow & \text{`}final\ PairCC\ p = \\ new\ Pair(a, b);' \ P) & & new\ PairCC(a, b)' \ J(P) \\ & & \\ J(\text{`}final\ Type\ a = p.getLeft();' \ P) & \rightarrow & \text{`}finalTypeCC\ a = \\ & & (TypeCC)p.getLeft();' \ J(P) \\ & & \\ J(\text{`}final\ Type\ b = p.getRight();' \ P) & \rightarrow & \text{`}finalTypeCC\ b = \\ & & (TypeCC)p.getRight();' \ J(P) \\ & & \\ J(\text{`}fail();') & \rightarrow & \text{`}throw\ new \\ & & SpiWrapperException(\text{"}checkfailed\text{"});' \\ J(\text{`}event(\text{"}name\text{"}, params);' \ P) & \rightarrow & J(P) \end{array}$$

Translation of statements regarding cryptographic operations. In this case some additional tokens are defined, for instance “SubtypeHashing” is a token that can be translated into “CryptoHashing” or “DHHashing”, depending on annotations put on the JavaSPI code.

$$\begin{array}{rcl} J(\text{`}final\ Hashing\ h = & \rightarrow & \text{`}final\ HashingCC\ h = \\ new\ Hashing(a);' \ P) & & new\ SubtypeHashingCC \\ & & (a, params);' \ J(P) \end{array}$$

$$
\begin{array}{rcl}
J(\text{`}final\ SharedKey\ sk = new\ SharedKey(a);'\ P) & \rightarrow & \text{`}final\ SharedKeyCC\ sk = new\ SharedKeyCC(a, params);'\ J(P) \\
J(\text{`}final\ KeyPair\ kp = new\ KeyPair(a);'\ P) & \rightarrow & \text{`}final\ KeyPair\ kp = new\ KeyPair(a, params);'\ J(P) \\
J(\text{`}final\ SharedKeyCiphered < Type > x = new\ SharedKeyCiphered < Type > (k, m)'\ P) & \rightarrow & \text{`}final\ SharedKeyCipheredCC\ x = new\ SharedKeyCipheredCC\ (k, m, params)'\ J(P) \\
J(\text{`}final\ PrivateKeyCiphered < Type > x = new\ PrivateKeyCiphered < Type\ (k, m)'\ P) & \rightarrow & \text{`}final\ PrivateKeyCipheredCC\ x = new\ PrivateKeyCipheredCC\ (k, m, params)'\ J(P) \\
J(\text{`}final\ PublicKeyCiphered < Type > x = new\ PublicKeyCiphered < Type > (k, m)'\ P) & \rightarrow & \text{`}final\ PublicKeyCipheredCC\ x = new\ PublicKeyCipheredCC\ (k, m, params)'\ J(P) \\
J(\text{`}final\ Type\ t = x.decrypt(k);'\ P) & \rightarrow & \text{`}final\ TypeCC\ t = (TypeCC)x.decrypt(k)'\ J(P) \\
J(\text{`}final\ Type\ t = x.decrypt\_w(k);'\ P) & \rightarrow & \text{`}final\ TypeCC\ t = (TypeCC)x.decrypt\_w(k)'\ J(P) \\
J(\text{`}final\ Type\ t = rc.getValue();'\ P) & \rightarrow & \text{`}final\ TypeCC\ t = (TypeCC)rc.getValue();'\ J(P)
\end{array}
$$

Translation of statements regarding generic conditions and data transmissions

$$
\begin{array}{rcl}
J(\text{`}if(cond)\{'\ P\ \text{`}\}') & \rightarrow & \text{`}if(cond)\{'J(P)\text{`}\}' \\
J(\text{`}if(cond)\{'\ P\ \text{`}\}else\{'\ Q\ \text{`}\}') & \rightarrow & \text{`}if(cond)\{'J(P)\text{`}\}else\{'J(Q)\text{`}\}' \\
J(\text{`}cAB.send(m);'\ P) & \rightarrow & \text{`}cAB.send(m);'\ J(P) \\
J(\text{`}final\ Type\ t = cAB.receive(Type.class);'\ P) & \rightarrow & \text{`}final\ TypeCC\ t = cAB.receive(new\ TypeCC());'\ J(P)
\end{array}
$$

A.4 $PV()$ translation rules

Under the assumption that $PV(\sigma) = \sigma$ translation rules are defined only on P, while σ have been omitted.

Translation of statements regarding packet composition / decomposition and other accessory rules:

$$
\begin{array}{rcl}
PV(\text{`}final\ Type\ t = new\ Type(data);'\ P) & \rightarrow & \text{`}new\ t;'\ PV(P)
\end{array}
$$

$$PV(\text{`}final\ Pair < A, B > p = new\ Pair(a,b);'\ P) \rightarrow \text{`}let\ p = (a,b)\ in'\ PV(P)$$

$$PV(\text{`}final\ Integer\ s = new\ Integer(x);'\ P) \rightarrow \text{`}let\ s = suc(x)\ in'\ PV(P)$$

$$PV(\text{`}final\ Type\ a = p.getLeft();'\ P) \rightarrow \text{`}let\ a = GetLeft(p)\ in'\ PV(P)$$

$$PV(\text{`}final\ Type\ b = p.getRight();'\ P) \rightarrow \text{`}let\ a = GetRight(p)\ in'\ PV(P)$$

$$PV(\text{`}fail();') \rightarrow \text{`}'$$

$$PV(\text{`}event(\text{"}name\text{"}, params);'\ P) \rightarrow \text{`}event\ evt\_name((params));'$$

Translation of statements regarding cryptographic operations

$$PV(\text{`}final\ Hashing\ h = new\ Hashing(a);'\ P) \rightarrow \text{`}let\ h = H(a)\ in'\ PV(P)$$

$$PV(\text{`}final\ SharedKey\ sk = new\ SharedKey(a);'\ P) \rightarrow \text{`}let\ sk = SharedKey(a) in'\ PV(P)$$

$$PV(\text{`}final\ KeyPair\ kp = new\ KeyPair(a);'\ P) \rightarrow \text{`}let\ kp = (PubPart(a), PriPart(a))\ in'\ PV(P)$$

$$PV(\text{`}final\ SharedKeyCiphered < Type > x = new\ SharedKeyCiphered(k,m)'\ P) \rightarrow \text{`}let\ x = SymEncrypt(k,m) in'\ PV(P)$$

$$PV(\text{`}final\ PrivateKeyCiphered < Type > x = new\ PrivateKeyCiphered(k,m)'\ P) \rightarrow \text{`}let\ x = PriEncrypt(k,m) in'\ PV(P)$$

$$PV(\text{`}final\ PublicKeyCiphered < Type > x = new\ PublicKeyCiphered(k,m)'\ P) \rightarrow \text{`}let\ x = PubEncrypt(k,m) in'\ PV(P)$$

$$PV(\text{`}final\ Type\ t = x.decrypt(k);'\ P), \{x \rightarrow \{m\}'_k\} \in \delta \rightarrow \text{`}let\ t = SymDecrypt(k,x) in'\ PV(P)$$

$$PV(\text{`}final\ Type\ t = x.decrypt(k);'\ P), \{x \rightarrow \{[m]\}'_k \in \delta \rightarrow \text{`}let\ t = PubDecrypt(k,x) in'\ PV(P)$$

$$PV(\text{`}final\ Type\ t = x.decrypt(k);'\ P), \{x \rightarrow [\{m\}]'_k \in \delta \rightarrow \text{`}let\ t = PriDecrypt(k,x) in'\ PV(P)$$

$$PV(\text{`}final\ RC\ rc = x.decrypt\_w(k);' \text{`}if(rc.isValid()\{' \text{`}final\ Type\ t = rc.getValue();'\ P\ \text{`}\}else\{'Q\text{`}\}') \{x \rightarrow \{m\}'_k \in \delta \rightarrow \text{`}let\ t = SymDecrypt(k,x) in' \text{`}('PV(P)\text{`})else('PV(Q)\text{`})'$$

$$PV(\text{`}final\ Type\ t = x.decrypt(k);' \text{`}if(rc.isValid()\{' \text{`}final\ Type\ t = rc.getValue();'\ P\ \text{`}\}else\{'Q\text{`}\}') \{x \rightarrow \{[m]\}'_k \in \delta \rightarrow \text{`}let\ t = PubDecrypt(k,x) in' \text{`}('PV(P)\text{`})else('PV(Q)\text{`})'$$

$$PV(\text{`}final\ Type\ t = x.decrypt(k);' \text{`}if(rc.isValid()\{' \rightarrow \text{`}let\ t = PriDecrypt(k,x) in' \text{`}('PV(P)\text{`})else('PV(Q)\text{`})'$$

$$\begin{array}{r} `final\ Type\ t = rc.getValue();' P \\ `\}else\{' Q `\}') \\ \{x \to [\{m\}]'_k \in \delta \end{array}$$

Translation of statements regarding generic conditions and data transmissions

$PV(`if(a.equals(b))\{'\ P\ `\}')$	$\to$	$`if\ a = b\ then('\ PV(P)\ `)'$
$PV(`if(a.equals(b))\{'\ P\ `\}else\{'\ Q\ `\}')$	$\to$	$`if\ a = b\ then('\ PV(P)\ `)else('\ PV(Q)\ `)'$
$PV(`cAB.send(m);'\ P)$	$\to$	$`out(cAB, m);'\ PV(P)$
$PV(`final\ Type\ t = cAB.receive(Type.class);'\ P)$	$\to$	$`in(cAB, m);'\ PV(P)$